

LINEAR-PROGRAMMING DESIGN AND ANALYSIS OF FAST ALGORITHMS FOR MAX 2-CSP

ALEXANDER D. SCOTT* AND GREGORY B. SORKIN

ABSTRACT. The class Max $(r, 2)$ -CSP, or simply Max 2-CSP, consists of constraint satisfaction problems with at most two r -valued variables per clause. For instances with n variables and m binary clauses, we present an $O(nr^{5+19m/100})$ -time algorithm which is the fastest polynomial-space algorithm for many problems in the class, including Max Cut. The method also proves a treewidth bound $\text{tw}(G) \leq (13/75 + o(1))m$, which gives a faster Max 2-CSP algorithm that uses exponential space: running in time $O^*(2^{(13/75+o(1))m})$, this is fastest for most problems in Max 2-CSP. Parametrizing in terms of n rather than m , for graphs of average degree d we show a simple algorithm running time $O^*(2^{(1-\frac{2}{d+1})n})$, the fastest polynomial-space algorithm known.

In combination with “Polynomial CSPs” introduced in a companion paper, these algorithms also allow (with an additional polynomial-factor overhead in space and time) counting and sampling, and the solution of problems like Max Bisection that escape the usual CSP framework.

Linear programming is key to the design as well as the analysis of the algorithms.

1. INTRODUCTION

A recent line of research has been to speed up exponential-time algorithms for sparse instances of maximization problems such as Max 2-Sat and Max Cut. The typical method is to repeatedly *transform* an instance to a smaller one or *split* it into several smaller ones (whence the exponential running time) until trivial instances are reached; the reductions are then reversed to recover a solution to the original instance. In [SS03] we introduced a new such method, distinguished by the fact that reducing an instance of Max Cut, for example, results in a problem that may not belong to Max Cut, but where the reductions are closed over a larger class, Max 2-CSP, of constraint satisfaction problems with at most two variables per clause. This allowed the reductions to be simpler, fewer, and more powerful. The algorithm ran in time $O^*(2^{m/5})$ (time $O^*(r^{m/5})$ for r -valued problems), making it the fastest for Max Cut, and tied (at the time) for Max 2-Sat.

In this paper we present a variety of results on faster exponential-time CSP algorithms and on treewidth. Our approach uses linear programming in both the design and the analysis of the algorithms.

1.1. Results. The running times for our algorithms depend on the space allowed, and are summarized in Table 1. The $O^*(\cdot)$ notation, which ignores leading polynomial factors, is defined in Section 2.1.

For Max 2-CSP we give an $O^*(r^{19m/100})$ -time, linear-space algorithm. This is the fastest polynomial-space algorithm known for Max Cut, Max Dicut, Max 2-Lin, less common problems such as Max Ones 2-Sat, weighted versions of all these, and of course general Max 2-CSP; more efficient algorithms are known for only a few problems, such as Maximum Independent Set and Max 2-Sat. If exponential space is allowed, we give an algorithm running in time $O^*(r^{(13/75+o(1))m})$ and space $O^*(r^{(1/9+o(1))m})$; it is the fastest exponential-space algorithm known for most problems in Max 2-CSP (including those listed above for the polynomial-space algorithm).

Key words and phrases. Max Cut; Max 2-Sat; Max 2-CSP; exact algorithms; linear-programming duality; measure and conquer;

* Research supported in part by EPSRC grant GR/S26323/01.

edge parametrized (m)					
problem	time (exact)	time (numerical)		space	reference
Max $(r, 2)$ -CSP	$O^*(r^{19m/100})$	$O^*(r^{0.19m})$	$O^*(r^{m/5.2631})$	linear	Theorem 11
$\Delta \leq 4$	$O^*(r^{3m/16})$	$O^*(r^{0.1875m})$	$O^*(r^{m/5.3333})$		
$\Delta \leq 3$	$O^*(r^{m/6})$	$O^*(r^{0.1677m})$	$O^*(r^{m/6})$		
Max $(r, 2)$ -CSP	$O^*(r^{(13/75+o(1))m})$	$O^*(r^{0.1734m})$	$O^*(r^{m/5.7692})$	exponential	Corollary 23
$\Delta \leq 4$	$O^*(r^{(1/6+o(1))m})$	$O^*(r^{0.1667m})$	$O^*(r^{m/5.9999})$		
$\Delta \leq 3$	$O^*(r^{(1/9+o(1))m})$	$O^*(r^{0.1112m})$	$O^*(r^{m/8.9999})$		
vertex parametrized (n)					
Max $(r, 2)$ -CSP	$O^*(r^{(1-\frac{2}{d+1})n})$			polynomial	Theorem 13
	$O^*(r^{(d-2)n/4})$			polynomial	[SS03, SS06c]

TABLE 1. Exact bounds and numerical bounds (in two forms) on the running times of our Max $(r, 2)$ -CSP algorithms. All of these are the best known. Throughout this paper, m denotes the number of 2-clauses and n the number of variables; Δ denotes the maximum number of 2-clauses on any variable, and d the average number.

These bounds have connections with treewidth, and we prove that the treewidth of an m -edge graph G satisfies $\text{tw}(G) \leq 3 + 19m/100$ and $\text{tw}(G) \leq (13/75 + o(1))m$. (The second bound is clearly better for large m .)

For both treewidth and algorithm running time we provide slightly better results for graphs of maximum degree $\Delta(G) = 3$ and $\Delta(G) = 4$.

In combination with a “Polynomial CSP” approach presented in a companion paper [SS06a, SS07], the algorithms here also enable (with an additional polynomial-factor overhead in space and time) counting CSP solutions of each possible cost; sampling uniformly from optimal solutions; sampling from all solutions according to the Gibbs measure or other distributions; and solving problems that do not fall into the Max 2-CSP framework, like Max Bisection, Sparsest Cut, judicious partitioning, Max Clique (without blowing up the input size), and multi-objective problems. We refer to [SS06a, SS07] for further details.

Our emphasis is on running time parametrized in terms of the number of edges m , but we also have results for parametrization in terms of the number of edges n (obtained largely independently of the methods in the rest of the paper). The main new result is a Max 2-CSP algorithm running in time $O^*(r^{(1-\frac{2}{d+1})n})$ (Theorem 13), where d is the average number of appearances of each variable in 2-clauses. Coupled with an older algorithm of ours (see [SS03, SS06c]) with running time $O^*(r^{(1-\frac{2}{d+2})n})$, this is the best known polynomial-space algorithm.

1.2. Techniques. We focus throughout on the “constraint graph” supporting a CSP instance. Our algorithms use several simple transformation rules, and a single splitting rule. The transformation rules replace an instance by an equivalent instance with fewer variables; our splitting rule produces several instances, each with the same, smaller, constraint graph. In a simple recursive CSP algorithm, then, the size of the CSP “recursion tree” is exponential in the number of splitting reductions on the graph. The key step in the analysis of our earlier $O^*(r^{m/5})$ algorithm was to show that the number of splitting reductions for an m -edge graph can be no more than $m/5$.

We used a linear programming (LP) analysis to derive an upper bound on how large the number of splitting reductions can be. Each reduction affects the degree sequence of the graph in a simple

way, and the fact that the number of vertices of each degree is originally non-negative and finally 0 is enough to derive the $m/5$ bound.

It is not possible to improve upon the $m/5$ bound on the *number* of splitting reductions, since there are examples achieving the bound. However, we are able to obtain a smaller bound on the reduction “depth” (described later), and the running time of a more sophisticated algorithm is exponential in this depth. Analysis of the reduction depth must take into account the component structure of the CSP’s constraint graph. The component structure is not naturally captured by the LP analysis, which considers the (indivisible) degree sequence of the full graph (the usual argument that in case of component division “we are done, by induction” cannot be applied) but a slight modification of the argument resolves the difficulty.

We note that the LP was essential in the *design* of the new algorithm as well as its analysis. The support of the LP’s *primal* solution indicates the particular reductions that contribute to the worst case. With a “bad” reduction identified, we do two things in parallel: exclude the reduction from the LP to see if an improved bound would result, and apply some actual thinking to see if it is possible to avoid the bad reduction. Since thinking is difficult and time-consuming, it is nice that the LP can be modified and re-run in a second to determine whether any gain would actually result. Furthermore, the LP’s *dual* solution gives an (optimal) set of weights, for edges and for vertices of each degree, for a “Lyapunov” or “potential function” proof of the depth bound. The potential-function proof method is well established (in the exponential-time algorithm context the survey [FGK05] calls it a “measure and conquer” strategy), and the LP method gives an efficient and provably optimal way of carrying it out.

The LP method presented is certainly applicable to reductions other than our own, and we also hope to see it applied to algorithm design and analysis in contexts other than exponential-time algorithms and CSPs. (For a different use of LPs in automating extremal constructions, see [TSSW00].)

1.3. Literature survey. We are not sure where the class (a, b) -CSP was first introduced, but this model, where each variable has at most a possible colors and there are general constraints each involving at most b variables, is extensively exploited for example in Beigel and Eppstein’s $O^*(1.3829^n)$ -time 3-coloring algorithm [BE05]. Finding relatively fast exponential-time algorithms for NP-hard problems is a field of endeavor that includes Schöning’s famous randomized algorithm for 3-Sat [Sch99], taking time $O^*((4/3)^n)$ for an instance on n variables.

Narrowing the scope to Max 2-CSPs with time parametrized in m , we begin our history with an algorithm of Niedermeier and Rossmanith [NR00]: designed for Max Sat generally, it solves Max 2-Sat instances in time $O^*(2^{0.348m})$. The Max 2-Sat result was improved by Hirsch to $O^*(2^{m/4})$ [Hir00]. Gramm, Hirsch, Niedermeier and Rossmanith showed how to solve Max 2-Sat in time $O^*(2^{m/5})$, and used a transformation from Max Cut into Max 2-Sat to allow Max Cut’s solution in time $O^*(2^{m/3})$ [GHN03]. Kulikov and Fedin showed how to solve Max Cut in time $O^*(2^{m/4})$ [KF02]. Our own [SS03] improved the Max Cut time (and any Max 2-CSP) to $O^*(2^{m/5})$. Kojevnikov and Kulikov recently improved the Max 2-Sat time to $O^*(2^{m/5.5})$ [KK06]; at the time of writing this is the fastest.

We now improve the time for Max Cut to $O^*(2^{19m/100})$. We also give linear-space algorithms for all of Max 2-CSP running in time $O^*(r^{19m/100})$, as well as faster but exponential-space algorithms running in time $O^*(2^{(13/75+o(1))m})$. and space $O^*(2^{(1/9+o(1))m})$. All these new results are the best currently known.

A technical report of Kneis and Rossmanith [KR05] (published just months after our [SS04]), and a subsequent paper of Kneis, Mölle, Richter and Rossmanith [KMRR05], give results overlapping with those in [SS04] and the present paper. They give algorithms applying to several problems in Max 2-CSP, with claimed running times of $O^*(2^{19m/100})$ and (in exponential space) $O^*(2^{13m/75})$. The papers are widely cited but confuse the literature to a degree. First, the authors were evidently

unaware of [SS04]. [KR05] cites our much earlier conference paper [SS03] (which introduced many of the ideas extended in [SS04] and the present paper) but overlooks both its $O^*(2^{m/5})$ algorithm and its II-reduction (which would have extended their results to all of Max 2-CSP). These oversights are repeated in [KMRR05]. Also, both papers have a reparable but fairly serious flaw, as they overlook the “component-splitting” case C4 of Section 5.4 (see Section 5 below). Rectifying this means adding the missing case, modifying the algorithm to work component-wise, and analyzing “III-reduction depth” rather than the total number of III-reductions — the issues that occupy us throughout Section 5. While treewidth-based algorithms have a substantial history (surveyed briefly in Section 7), [KR05] and [KMRR05] motivate our own exploration of treewidth, especially Subsection 7.2’s use of Fomin and Høie’s [FH06].

We turn our attention briefly to algorithms parametrized in terms of the number n of vertices (or variables), along with the average degree d (the average number of appearances of a variable in 2-clauses) and the maximum degree Δ . A recent result of Della Croce, Kaminski, and Paschos [DCKP] solves Max Cut (specifically) in time $O^*(2^{mn/(m+n)}) = O^*(2^{(1-\frac{2}{d+2})n})$. Another recent paper, of Fürer and Kasiviswanathan [FK07], gives a running-time bound of $O^*(2^{(1-\frac{1}{d-1})n})$ for any Max 2-CSP (where $d > 2$ and the constraint graph is connected, per personal communication). Both of these results are superseded by the Max 2-CSP algorithm of Theorem 13, with time bound $O^*(2^{(1-\frac{2}{d+1})n})$, coupled with another of our algorithms from [SS03, SS06c], with running time $O^*(2^{(1-\frac{2}{d+2})n})$. A second algorithm from [DCKP], solving Max Cut in time $O^*(2^{(1-2/\Delta)n})$, remains best for “nearly regular” instances where $\Delta \leq d + 1$.

Particular problems within Max 2-CSP can often be solved faster. For example, an easy tailoring of our $O^*(r^{19m/100})$ algorithm to weighted Maximum Independent Set runs in time $O^*(2^{3n/8})$ (see Corollary 14), which is $O^*(1.2969^n)$. This improves upon an older algorithm of Dahllöf and Jonsson [DJ02], but is not as good as the $O^*(1.2561^n)$ algorithm of Dahllöf, Jonsson and Wahlström [DJW05] or the $O^*(1.2461^n)$ algorithm of Fürer and Kasiviswanathan [FK05]. (Even faster algorithms are known for unweighted MIS.)

The elegant algorithm of Williams [Wil04], like our algorithms, applies to all of Max 2-CSP. It is the only known algorithm to treat *dense* instances of Max 2-CSP relatively efficiently, and also enjoys some of the strengths of our Polynomial CSP extension [SS06a, SS07]. It intrinsically requires exponential space, of order $2^{2n/3}$, and runs in time $O^*(2^{\omega n/3})$, where $\omega < 2.376$ is the matrix-multiplication exponent. Noting the dependency on n rather than m , this algorithm is faster than our polynomial-space algorithm if the average degree is above $2(\omega/3)/(19/100) < 8.337$, and faster than our exponential-space algorithm if the average degree is above 9.139.

An early version of our results was given in the technical report [SS04], and a conference version appeared as [SS06b].

1.4. Outline. In the next section we define the class Max 2-CSP, and in Section 3 we introduce the reductions our algorithms will use. In Section 4 we define and analyze the $O(nr^{3+m/5})$ algorithm of [SS03] as a relatively gentle introduction to the tools, including the LP analysis. The $O(nr^{5+19m/100})$ algorithm is presented in Section 5; it entails a new focus on components of the constraint graph, affecting the algorithm and the analysis. Section 6 digresses to consider algorithms with run time parametrized by the number of vertices rather than edges; by this measure, it gives the fastest known polynomial-space algorithm for general Max 2-CSP instances. Section 7 presents corollaries pertaining to the treewidth of a graph and the exponential-space $O^*(r^{(13/75+o(1))m})$ algorithm. Section 8 recapitulates, and considers the potential for extending the approach in various ways.

2. MAX $(r, 2)$ -CSP

The problem Max Cut is to partition the vertices of a given graph into two classes so as to maximize the number of edges “cut” by the partition. Think of each *edge* as being a *function* on

the classes (or “colors”) of its endpoints, with value 1 if the endpoints are of different colors, 0 if they are the same: Max Cut is equivalent to finding a 2-coloring of the vertices which maximizes the sum of these edge functions. This view naturally suggests a generalization.

An *instance* (G, S) of Max $(r, 2)$ -CSP is given by a “constraint” graph $G = (V, E)$ and a set S of “score” functions. Writing $[r] = \{1, \dots, r\}$ for the set of available colors, we have a “dyadic” score function $s_e : [r]^2 \rightarrow \mathbb{R}$ for each edge $e \in E$, a “monadic” score function $s_v : [r] \rightarrow \mathbb{R}$ for each vertex $v \in V$, and finally a single “niladic” score “function” $s_\emptyset : [r]^0 \rightarrow \mathbb{R}$ which takes no arguments and is just a constant convenient for bookkeeping.

A *candidate solution* is a function $\phi : V \rightarrow [r]$ assigning “colors” to the vertices (we call ϕ an “assignment” or “coloring”), and its *score* is

$$s(\phi) := s_\emptyset + \sum_{v \in V} s_v(\phi(v)) + \sum_{uv \in E} s_{uv}(\phi(u), \phi(v)). \quad (1)$$

An *optimal solution* ϕ is one which maximizes $s(\phi)$.

We don’t want to belabor the notation for edges, but we wish to take each edge just once, and (since s_{uv} need not be a symmetric function) with a fixed notion of which endpoint is “ u ” and which is “ v ”. We will typically assume that $V = [n]$ and any edge uv is really an ordered pair (u, v) with $1 \leq u < v \leq n$; we will also feel free to abbreviate $s_{uv}(C, D)$ as $s_{uv}(CD)$, etc.

Henceforth we will simply write Max 2-CSP for the class Max $(r, 2)$ -CSP. The “2” here refers to score functions’ taking 2 or fewer arguments: 3-Sat, for example, is out of scope. Replacing 2 by a larger value would mean replacing the constraint graph with a hypergraph, and changes the picture significantly.

An obvious computational-complexity issue is raised by our allowing scores to be arbitrary *real* values. Our algorithms will add, subtract, and compare these scores, never introducing a number larger in absolute value than the sum of the absolute values of all input values, and we assume that each such operation can be done in time and space $O(1)$. If desired, scores may be limited to integers, and the length of the integers factored in to the algorithm’s complexity, but this seems uninteresting and we will not remark on it further.

2.1. Notation. We reserve the symbols G for the constraint graph of a Max 2-CSP instance, n and m for its numbers of vertices and edges, $[r] = \{1, \dots, r\}$ for the allowed colors of each vertex, and $L = 1 + nr + mr^2$ for the input length. Since a CSP instance with $r < 2$ is trivial, we will assume $r \geq 2$ as part of the definition.

For brevity, we will often write “ d -vertex” in lieu of “vertex of degree d ”. We write $\Delta(G)$ for the maximum degree of G .

The notation $O^*(\cdot)$ suppresses polynomial factors in any parameters, so for example $O^*(r^{cn})$ may mean $O(r^3 n r^{cn})$. To avoid any ambiguity in multivariate $O(\cdot)$ expressions, we take a strong interpretation that that $f(\cdot) = O(g(\cdot))$ if there exists some constant C such that $f(\cdot) \leq Cg(\cdot)$ for *all* values of their (common) arguments. (To avoid some notational awkwardness, we disallow the case $n = 0$, but allow $m = 0$.)

2.2. Remarks. Our assumption of an undirected constraint graph is sound even for a problem such as Max Dicut (maximum directed cut). For example, for Max Dicut a directed edge (u, v) with $u < v$ would be expressed by the score function $s_{uv}(\phi(u), \phi(v)) = 1$ if $(\phi(u), \phi(v)) = (0, 1)$ and $s_{uv}(\phi(u), \phi(v)) = 0$ otherwise; symmetrically, a directed edge (v, u) , again with $u < v$, would have score $s_{uv}(\phi(u), \phi(v)) = 1$ if $(\phi(u), \phi(v)) = (1, 0)$ and score 0 otherwise.

There is no loss of generality in assuming that an input instance has a simple constraint graph (no loops or multiple edges), or by considering only maximization and not minimization problems.

Readers familiar with the class \mathcal{F} -Sat (see for example Marx [Mar04], Creignou [Cre95], or Khanna [KSTW01]) will realize that when the arity of \mathcal{F} is limited to 2, Max 2-CSP contains \mathcal{F} -Sat, \mathcal{F} -Max-Sat and \mathcal{F} -Min-Sat; this includes Max 2-Sat and Max 2-Lin (satisfying as many as

possible of m 2-variable linear equalities and/or inequalities). Max 2-CSP also contains \mathcal{F} -Max-Ones; for example Max-Ones-2-Sat. Additionally, Max 2-CSP contains similar problems where we maximize the weight rather than merely the number of satisfied clauses.

The class Max 2-CSP is surprisingly flexible, and in addition to Max Cut and Max 2-Sat includes problems like MIS and minimum vertex cover that are not at first inspection structured around pairwise constraints. For instance, to model MIS as a Max 2-CSP, let $\phi(v) = 1$ if vertex v is to be included in the independent set, and 0 otherwise; define vertex scores $s_v(\phi(v)) = \phi(v)$; and define edge scores $s_{uv}(\phi(u), \phi(v)) = -2$ if $\phi(u) = \phi(v) = 1$, and 0 otherwise.

3. REDUCTIONS

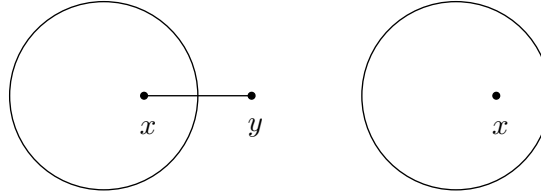
As with most of the works surveyed above, our algorithms are based on progressively reducing an instance to one with fewer vertices and edges until the instance becomes trivial. Because we work in the general class Max 2-CSP rather than trying to stay within a smaller class such as Max 2-Sat or Max k -Cut, our reductions are simpler and fewer than is typical. For example, [GHN03] uses seven reduction rules; we have just three (plus a trivial “0-reduction” that other works may treat implicitly). The first two reductions each produce equivalent instances with one vertex fewer, while the third produces a set of r instances, each with one vertex fewer, some one of which is equivalent to the original instance. We expand the previous notation (G, S) for an instance to (V, E, S) , where $G = (V, E)$.

Reduction 0 (transformation): This is a trivial “pseudo-reduction”. If a vertex y has degree 0 (so it has no dyadic constraints), then set $s_\emptyset = s_\emptyset + \max_{C \in [r]} s_y(C)$ and delete y from the instance entirely.

Reduction I: Let y be a vertex of degree 1, with neighbor x . Reducing (V, E, S) on y results in a new problem (V', E', S') with $V' = V \setminus y$ and $E' = E \setminus xy$. S' is the restriction of S to V' and E' , except that for all colors $C \in [r]$ we set

$$s'_x(C) = s_x(C) + \max_{D \in [r]} \{s_{xy}(CD) + s_y(D)\}.$$

Note that any coloring ϕ' of V' can be extended to a coloring ϕ of V in r ways, depending on the color assigned to y . Writing (ϕ', D) for the extension in which $\phi(y) = D$, the defining property of the reduction is that $s'(\phi') = \max_D s(\phi', D)$. In particular, $\max_{\phi'} s'(\phi') = \max_{\phi} s(\phi)$, and an optimal coloring ϕ' for the instance (V', E', S') can be extended to an optimal coloring ϕ for (V, E, S) .



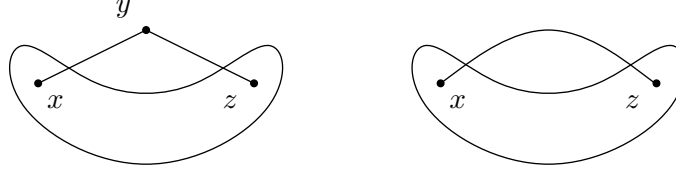
Reduction II (transformation): Let y be a vertex of degree 2, with neighbors x and z . Reducing (V, E, S) on y results in a new problem (V', E', S') with $V' = V \setminus y$ and $E' = (E \setminus \{xy, yz\}) \cup \{xz\}$. S' is the restriction of S to V' and E' , except that for $C, D \in [r]$ we set

$$s'_{xz}(CD) = s_{xz}(CD) + \max_{F \in [r]} \{s_{xy}(CF) + s_{yz}(FD) + s_y(F)\} \quad (2)$$

if there was already an edge xz , discarding the first term $s_{xz}(CD)$ if there was not.

As in Reduction I, any coloring ϕ' of V' can be extended to V in r ways, according to the color F assigned to y , and the defining property of the reduction is that $s'(\phi') =$

$\max_F s(\phi', F)$. In particular, $\max_{\phi'} s'(\phi') = \max_{\phi} s(\phi)$, and an optimal coloring ϕ' for (V', E', S') can be extended to an optimal coloring ϕ for (V, E, S) .



Reduction III (splitting): Let y be a vertex of degree 3 or higher. Where reductions I and II each had a single reduction of (V, E, S) to (V', E', S') , here we define r different reductions: for each color C there is a reduction of (V, E, S) to (V', E', S^C) corresponding to assigning the color C to y . We define $V' = V \setminus y$, and E' as the restriction of E to $V \setminus y$. S^C is the restriction of S to $V \setminus y$, except that we set

$$(s^C)_0 = s_0 + s_y(C),$$

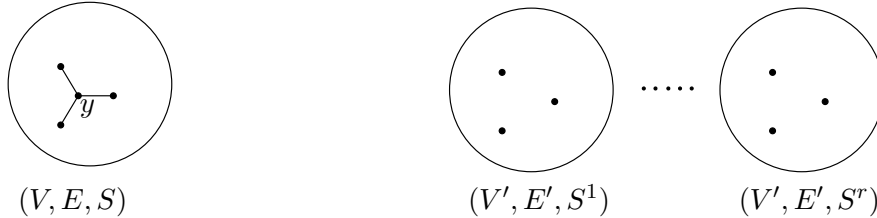
and, for every neighbor x of y and every $D \in [r]$,

$$(s^C)_x(D) = s_x(D) + s_{xy}(DC).$$

As in the previous reductions, any coloring ϕ' of $V \setminus y$ can be extended to V in r ways: for each color C there is an extension (ϕ', C) , where color C is given to y . We then have (this is different!) $s^C(\phi') = s(\phi', C)$, and furthermore,

$$\max_C \max_{\phi'} s^C(\phi') = \max_{\phi} s(\phi),$$

where an optimal coloring on the left is an optimal coloring on the right.



Note that each of the reductions above has a well-defined effect on the constraint graph of an instance: A 0-reduction deletes its (isolated) vertex; a I-reduction deletes its vertex (of degree 1); a II-reduction contracts away its vertex (of degree 2); and a III-reduction deletes its vertex (of degree 3 or more), independent of the “color” of the reduction. That is, all the CSP reductions have graph-reduction counterparts depending only on the constraint graph and the reduction vertex.

4. AN $O(nr^{3+m/5})$ ALGORITHM

As a warm-up to our $O^*(r^{19m/100})$ algorithm, in this section we will present Algorithm A, which will run in time $O(nr^{3+m/5})$ and space $O(L)$. (Recall that $L = 1 + nr + mr^2$ is the input length.) Roughly speaking, a simple recursive algorithm for solving an input instance could work as follows. Begin with the input problem instance.

Given an instance $\mathcal{M} = (G, S)$:

- (1) If any reduction of type 0, I or II is possible (in that order of preference), apply it to reduce \mathcal{M} to \mathcal{M}' , recording certain information about the reduction. Solve \mathcal{M}' recursively, and use the recorded information to reverse the reduction and extend the solution to one for \mathcal{M} .

- (2) If only a type III reduction is possible, reduce (in order of preference) on a vertex of degree 5 or more, 4, or 3. For $i \in [r]$, recursively solve each of the instances \mathcal{M}^i in turn, select the solution with the largest score, and use the recorded information to reverse the reduction and extend the solution to one for \mathcal{M} .
- (3) If no reduction is possible then the graph has no vertices, there is a unique coloring (the empty coloring), and the score is s_\emptyset (from the niladic score function).

If the recursion depth — the number of III-reductions — is ℓ , the recursive algorithm's running time is $O^*(r^\ell)$. Thus in order to prove an $O^*(r^{m/5})$ bound on running time, it is enough to prove that $\ell \leq m/5$. We prove this bound in Lemma 4 in Section 4.6. (The preference order for type III reductions described above is needed to obtain the bound.)

In order to obtain our more precise $O(nr^{3+m/5})$ bound on running time, we must be a little more careful with the description of implementation and data storage. Thus Sections 4.1 to 4.5 deal with the additional difficulties arising from running in linear space and with a small polynomial factor for running time. A reader willing to take this for granted, or who is primarily interested in the exponent in the $O^*(r^{19m/100})$ bound, can skip directly to Section 4.6.

4.1. Linear space. If the recursion depth is ℓ , a straightforward recursive implementation would use greater-than-linear space, namely $\Theta(\ell L)$. Instead, when the algorithm has reduced on a vertex v , the reduced instance should be the only one maintained, while the pre-reduction instance should be reconstructible from compact ($O(1)$ -sized) information stored in the data structure for v .

4.2. Phases. For both efficiency of implementation and ease of analysis, we define Algorithm A as running in three phases. As noted at the end of Section 3, the CSP reductions have graph-reduction counterparts. In the first phase we merely perform such *graph* reductions. We reduce on vertices in the order of preference given earlier: 0-reduction (on a vertex of degree 0); I-reduction (on a vertex of degree 1); II-reduction (on a vertex of degree 2); or (still in order of preference) III-reduction on a vertex of degree 5 or more, 4, or 3. The output of this phase is simply the sequence of vertices on which we reduced.

The second phase finds the optimal *cost* recursively, following the reduction sequence of the first phase; if there were ℓ III-reductions in the first phase's reduction sequence, the second phase runs in time $O^*(r^\ell)$. The third phase is similar to the second phase and returns an optimal *coloring*.

4.3. First phase. In this subsection we show that a sequence of reductions following the stated preference order can be constructed in linear time and space by Algorithm A.1. (See displayed pseudocode, and details in Claim 1.)

Claim 1. *On input of a graph G with n vertices and m edges, Algorithm A.1 runs in time and space $O(m + n)$ and produces a reduction sequence obeying the stated preference order.*

Proof. Correctness of the algorithm is guaranteed by line 1. For the other steps we will have to detail some data structures and algorithmic details.

We assume a RAM model, so that a given memory location can be accessed in constant time. Let the input graph be presented in a sparse representation consisting of a vector of vertices, each with a doubly-linked list of incident edges, each edge with a pointer to the edge's twin copy indexed by the other endpoint. From the vector of vertices we create a doubly linked list of them, so that as vertices are removed from an instance to create a subinstance they are bridged over in the linked list, and there is always a linked list of just the vertices in the subinstance.

Transforming the input graph into a simple one can be done in time $O(m + n)$ and space $O(n)$. The procedure relies on a pointer array of length n , initially empty. For each vertex u , we iterate through the incident edges. For an edge to vertex v , if the v th entry of the pointer array is empty, we put a pointer to the edge uv . If the v th entry is not empty, this is not the first uv edge we have seen, and so we coalesce the new edge with the existing one: using the pointer to the original

Algorithm A.1: Algorithm A, first phase

```

1: Input a constraint graph  $G$ .
2: if  $G$  is not simple then
3:   Reduce it to a simple graph by identifying parallel edges.
4: end if
5: Sort the vertices into stacks, corresponding to degree 0, 1, 2,  $\geq 5$ , 4 and 3, in that order.
6: Let  $G_0 = G$ .
7: for  $i = 1$  to  $n$  do
8:   Pop a next-reduction vertex  $v_i$  from the first non-empty stack.
9:   if  $\deg(v_i) \geq 5$  then
10:    Check  $v_i$  for duplicate incident edges.
11:    Link any duplicate edge to the II-reduction that created it (using the label previously
    created by line 1).
12:  end if
13:  Reduce  $G_{i-1}$  on  $v_i$  to produce  $G_i$ , except:
14:  if  $v_i$  had degree 2 then
15:    Do not check whether the added edge duplicates an existing one; instead, label it as having
    been added by the reduction on  $v_i$ .
16:  end if
17:  Degree-check each  $G_{i-1}$ -neighbor of  $v_i$ .
18:  Place each neighbor on the appropriate stack, removing it from its old stack.
19: end for
20: Output the sequence  $v_1, \dots, v_n$  of reduction vertices, along with any duplicate-edge creations
    associated with each II-reduction vertex.

```

edge, we use the link from the redundant uv edge to its “ vu ” twin copy to delete the twin and bridge over it, then delete and bridge over the redundant uv edge itself. After processing the last edge for vertex u we run through its edges again, clearing the pointer array. The time to process a vertex u is of order the number of its incident edges (or $O(1)$ if it is isolated), so the total time is $O(m + n)$ as claimed. Henceforth we assume without loss of generality that the input instance has no multiple edges.

One of the trickier points is to maintain information about the degree of each vertex, because a II-reduction may introduce multiple edges and there is not time to run through its neighbors’ edges to detect and remove parallel edges immediately. However, it will be possible to track whether each vertex has degree 0, 1, 2, 3, 4, or 5 or more. We have a vertex “stack” for each of these cases. Each stack is maintained as a doubly linked list, and we keep pointers both ways between each vertex and its “marker” in the stack.

The stacks can easily be *created* in linear time from the input. The key to *maintaining* them is a *degree-checking* procedure for a vertex x . Iterate through x ’s incident edges, keeping track of the number of distinct neighboring vertices seen, stopping when we run out of edges or find 5 distinct neighbors. If a neighbor is repeated, coalesce the two edges. The time spent on x is $O(1)$ plus the number of edge coalescences. Once the degree of x is determined as 0, 1, 2, 3, 4, or 5 or more, remove x ’s marker from its old stack (using the link from x to delete the marker, and links from the marker to its predecessor and successor to bridge over it), and push a marker to x onto the appropriate new stack.

When reducing on vertex v , run the degree-checking procedure on each neighbor x of v (line 1 of Algorithm A.1). The time for this is the time to count up to 5 for each neighbor (a total of $O(\deg(v))$), plus the number of edge coalescences. Vertex degrees never increase above their initial values, so over the course of Algorithm A.1 the total of the $O(\deg(v))$ terms is $O(m)$.

Parallel edges are created only by II-reductions, each producing at most one such edge, so over the course of Algorithm A.1 at most n parallel edges are created, and the edge coalescences thus take time $O(n)$. The total time for degree-checking is therefore $O(m + n)$.

Finally, each reduction (line 1 of Algorithm A.1) can itself be performed in time $O(1 + \deg(v))$: for a 0, I, or III-reduction we simply delete v and its incident edges; for a II-reduction we do the same, then add an edge between v 's two former neighbors. Again, the total time is $O(m + n)$. \square

With Algorithm A's first phase Algorithm A.1 complete, we may assume henceforth that our graphs are always simple: from this phase's output we can (trivially) reproduce the sequence of reductions in time $O(m + n)$, and coalesce any duplicate edge the moment it appears.

4.4. Algorithm A: Second phase. The second phase, Algorithm A.2, determines the optimum cost, while the third and final phase, Algorithm A.3, returns a coloring with this cost. These two phases are nearly identical, and we proceed with Algorithm A.2.

Because the algorithm is recursive and limited to linear space, when recursing we cannot afford to pass a separate copy of the data; rather, a "subinstance" for recursion must be an in-place modification of the original data, and when a recursive call terminates it must restore the data to its original form. This recursion is sketched in Algorithm A.2 (see displayed pseudocode).

Algorithm A.2: Algorithm A, second phase recursively computing $s(G, S)$

```

1: Input: A CSP instance  $(G, S)$ , and reduction sequence  $\mathbf{v} := v_1, \dots, v_n$  (with associated
   duplicate-edge annotations, per Algorithm A.1 line 1).
2: if  $n = 0$  then
3:   Let  $s := s_\emptyset$ , the niladic score.
4:   Return  $(s, G, S, \mathbf{v})$ .
5: end if
6: if  $v_1$  is a 0-, I- or II-reduction vertex then
7:   Reduce  $(G, S)$  on  $v_1$  to obtain  $(G', S')$ 
8:   Record an  $O(r^2)$ -space annotation allowing the reduction on  $v_1$  to be reversed.
9:   Truncate the reduction sequence correspondingly, letting  $\mathbf{v}' := v_2, \dots, v_n$ .
10:  Let  $s := s(G', S')$ , computed recursively.
11:  Reverse the reduction to reconstruct  $(G, S)$  and  $\mathbf{v}$  (and free the storage from line 2).
12:  Return  $(s, G, S, \mathbf{v})$ .
13: else
14:    $v$  is a III-reduction vertex.
15:   Let  $s := -\infty$ .
16:   for color  $C = 1$  to  $r$  do
17:     III-reduce on  $v$  with color  $C$  to obtain  $(G', S^C)$ , and  $\mathbf{v}'$ .
18:     Record an  $O(\deg(v)r)$ -space reversal annotation.
19:     Let  $s := \max\{s, s(G', S^C)\}$ , computed recursively.
20:     Reverse the reduction to reconstruct  $(G, S)$  (and free the storage from line 2).
21:   end for
22:   Return  $(s, G, S, \mathbf{v})$ .
23: end if
24: Output:  $(s, G, S, \mathbf{v})$ , where  $s$  is the optimal score of  $(G, S)$ .
```

Claim 2. *Given an $(r, 2)$ -CSP instance with n vertices, m constraints, and length L , and a reduction sequence (per Algorithm A.1) with ℓ III-reductions, Algorithm A.2 returns the maximum score, using space $O(L)$ and time $O(r^{\ell+3}n)$.*

Proof. We first argue that each “branch” of the recursion (determined by the colors chosen in the III-reductions) requires space $O(L)$.

First we must detail how to implement the CSP reductions, which is a minor embellishment of the graph reduction implementations described earlier. Recall that there is a score function on each vertex, which we will assume is represented as an r -value table, and a similar function on each edge, represented as a table with r^2 values.

A CSP II-reduction on y with neighbors x and z follows the pattern of the graph reduction, but instead of simply constructing a new *edge* (x, z) we now construct a new *score function* s'_{xz} : iterate through all color pairs $C, D \in [r]$ and set $s'_{xz}(CD) := \max_{F \in [r]} \{s_{xy}(CF) + s_{yz}(FD) + s_y(F)\}$ as in (2). Iterating through values C, D and F takes time $O(r^3)$, and the resulting table takes space $O(r^2)$. If there already was a score function s_{xz} (if there already was an edge (x, z)), the new score function is the elementwise sum of the two tables. To reverse the reduction it suffices to record the neighbors x and z and keep around the old score functions s_{xy} and s_{yz} (allowing additional space $O(r^2)$ for the new one). Similarly, a I-reduction takes time $O(r^2)$ and space $O(r)$, and a 0-reduction time $O(r)$ and space $O(1)$.

To perform a III-reduction with color C on vertex y , for each neighbor x we incorporate the dyadic score $s_{yx}(CD)$ into the monadic score $s_x(D)$ (time $O(r)$ to iterate through $D \in [r]$), maintain for purposes of reversal the original score functions s_{yx} and s_x , and allocate space $O(r)$ for the new score function s'_x . Over all $\deg(y)$ neighbors the space required is $O(\deg(y)r)$, and for each of the r colors for the reduction, the time is also $O(\deg(y)r)$. (Note that $\deg(y) \neq 0$; indeed, $\deg(y) \geq 3$.)

Since vertex degrees are only decreased through the course of the algorithm, for one branch of the recursion the total space is $O(mr^2 + nr)$, i.e., $O(L)$. Since each branch of the recursion takes space $O(L)$, the same bound holds for the algorithm as a whole.

This concludes the analysis of space, and we turn to the running time. Let $f(n, \ell)$ be an upper bound on the running time for an instance with n nodes and III-recursion depth ℓ . We claim that $f(0, 0) = 1$ and for $n > 0$, $f(n, \ell) \leq r^3 n(r^\ell + (r^{\ell+1} - r)/(r - 1))$, presuming that we have “rescaled time” so that all absolute constants implicit in our $O(\cdot)$ expressions can be replaced by 1. (This is equivalent to claiming that for some sufficiently large absolute constant C , $f(0, 0) \leq C$ and $f(n, \ell) \leq Cr^3 n(r^\ell + (r^{\ell+1} - r)/(r - 1))$.) The case $n = 1$ is trivial. In the event of a recursive call in line (2), the recursion is preceded by just one 0-, I- or II-reduction, taking time $\leq r^3$; the other non-recursive steps may also be accounted for in the same r^3 time bound. By induction on n , in this case we have

$$\begin{aligned} f(n, \ell) &\leq r^3 + f(n - 1, \ell) \\ &\leq r^3 + r^3(n - 1)(r^\ell + (r^{\ell+1} - r)/(r - 1)) \\ &\leq r^3 n(r^\ell + (r^{\ell+1} - r)/(r - 1)), \end{aligned}$$

using only that $r^\ell + (r^{\ell+1} - r)/(r - 1) \geq r^\ell \geq 1$.

The interesting case is where there are r recursive calls originating in line (2), with the other lines in the loop (2) consuming time $O(r \cdot \deg(v)r)$; for convenience we bound this by $r^3 n$. In this case, by induction on n and ℓ ,

$$\begin{aligned} f(n, \ell) &\leq r^3 n + r f(n - 1, \ell - 1) \\ &\leq r^3 n + r \cdot r^3 n(r^{\ell-1} + (r^\ell - r)/(r - 1)) \\ &= r^3 n + r^3 n(r^\ell + (r^{\ell+1} - r^2)/(r - 1)) \\ &= r^3 n(r^\ell + (r^{\ell+1} - r^2 + r - 1)/(r - 1)) \\ &\leq r^3 n(r^\ell + (r^{\ell+1} - r)/(r - 1)), \end{aligned}$$

using $-r^2 + r - 1 \leq -r$ (from $0 \leq (r - 1)^2$). □

4.5. Algorithm A: Third phase. The third phase, Algorithm A.3 (not displayed) proceeds identically to the second until we visit a leaf achieving the maximum score (known from the second phase), at which point we backtrack through all the reductions, filling in the vertex colors.

There are two key points here. The first is that when a maximum-score leaf is hit, we know it, and can retrace back up the recursion tree. The second is the property that in retracing up the tree, when we reach a node v , all descendant nodes in the tree have been assigned optimal colors, v 's neighbors in the reduced graph correspond to such lower nodes, and thus we can optimally color v (recursively preserving the property). These points are obvious for Algorithm A.3 and so there is no need to write down its details, but we mention them because *neither property holds for Algorithm B*, whose third phase Algorithm B.3 is thus trickier.

Because Algorithm A.3 is basically just an interruption of Algorithm A.2 when a maximum-score leaf is encountered, the running time of Algorithm A.3 is no more than that of Algorithm A.2. We have thus established the following claim.

Claim 3. *Given an $(r, 2)$ -CSP instance with n vertices, m constraints, and length L , Algorithm A returns an optimal score and coloring in space $O(L)$ and time $O(r^{\ell+3}n)$, where ℓ is the number of III-reductions in the reduction sequence of Algorithm A.1.*

4.6. Recursion depth. The crux of the analysis is now to show that the number of III-reductions ℓ in the reduction sequence produced by Algorithm A's first phase is at most $m/5$.

Lemma 4. *Algorithm A.1 reduces a graph G with n vertices and m edges to a vertexless graph after no more than $m/5$ III-reductions.*

Proof. While the graph has maximum degree 5 or more, Algorithm A III-reduces only on such a vertex, destroying at least 5 edges; any I- or II-reductions only increase the number of edges destroyed. Thus, it suffices to prove the lemma for graphs with maximum degree 4 or less. Since the reductions never increase the degree of any vertex, the maximum degree will always remain at most 4.

In this paragraph, we give some intuition for the rest of the argument. Algorithm A III-reduces on vertices of degree 4 as long as possible, before III-reducing on vertices of degree 3, whose neighbors must then all be of degree 3 (vertices of degree 0, 1 or 2 would trigger a 0-, I- or II-reduction in preference to the III-reduction). Referring to Figure 1, note that each III-reduction on a vertex of degree 3 can be credited with destroying 6 edges, if we immediately follow up with II-reductions on its neighbors. (In Algorithm A we do not explicitly couple the II-reductions to the III-reduction, but the fact that the III-reduction creates 3 degree-2 vertices is sufficient to ensure the good outcome that intuition suggests. In Algorithm B we will have to make the coupling explicit.) Similarly, reduction on a 4-vertex destroys at least 5 edges unless the 4-vertex has no degree-3 neighbor. The only problem comes from reductions on vertices of degree 4 all of whose neighbors are also of degree 4, as these destroy only 4 edges. As we will see, the fact that such reductions also create 4 3-vertices, and the algorithm terminates with 0 3-vertices, is sufficient to limit the number of times they are performed.

We proceed by considering the various types of reduction and their effects on the number of edges and the number of 3-vertices. The reductions are catalogued in Table 2.

The first row, for example, shows that III-reducing on a vertex of degree 4 with 4 neighbors of degree 4 (and thus no neighbors of degree 3) destroys 4 edges, and (changing the neighbors from degree 4 to 3) destroys 5 vertices of degree 4 (including itself) and creates 4 vertices of degree 3. It counts as one III-reduction "step". The remaining rows up to the table's separating line similarly illustrate the other III-reductions. Below the line, II-reductions and I-reductions are decomposed into parts. As shown just below the line, a II-reduction, regardless of the degrees of the neighbors, first destroys 1 edge and 1 2-vertex, and counts as 0 steps (steps count only III-reductions). In the process, the II-reduction may create a parallel edge, which will promptly be deleted (coalesced) by

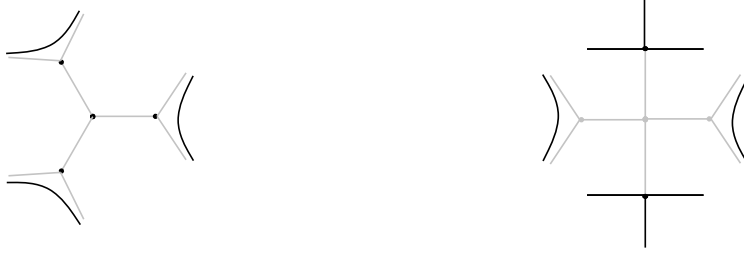


FIGURE 1. Left, a reduction on a 3-vertex with 3 3-neighbors, followed by II-reductions on those neighbors, destroys 6 edges and 4 3-vertices. (The original graph's edges are shown in grey and the reduced graph's edges in black.) Note that Algorithm A does not actually force any particular II-reductions after a III-reduction, but Algorithm B will do so. Right, a reduction on a 4-vertex with k 3-neighbors ($k = 2$ here) destroys $4 + k$ edges and $2k - 4$ 3-vertices (k 3-vertices are destroyed, but $4 - k$ 4-vertices become 3-vertices). The algorithm and analysis make no assumptions on the local structure of the graph; the figure is merely illustrative.

deg	#nbrs of deg				destroys					steps
	4	3	2	1	e	4	3	2	1	
4	4	0	0	0	4	5	-4	0	0	1
4	3	1	0	0	4	4	-2	-1	0	1
4	2	2	0	0	4	3	0	-2	0	1
4	1	3	0	0	4	2	2	-3	0	1
4	0	4	0	0	4	1	4	-4	0	1
3	0	3	0	0	3	0	4	-3	0	1
2					1	0	0	1	0	0
$\frac{1}{2}e$	1	0	0	0	$\frac{1}{2}$	1	-1	0	0	0
$\frac{1}{2}e$	0	1	0	0	$\frac{1}{2}$	0	1	-1	0	0
$\frac{1}{2}e$	0	0	1	0	$\frac{1}{2}$	0	0	1	-1	0
$\frac{1}{2}e$	0	0	0	1	$\frac{1}{2}$	0	0	0	1	0

TABLE 2. Tabulation of the effects of various reductions in Algorithm A.

Algorithm A. Since the exact effect of an edge deletion depends on the degrees of its neighbors, to minimize the number of cases we treat an edge deletion as two half-edge deletions, each of whose effects depends on the degree of the half-edge's incident vertex. For example the table's next line shows deletion of a half-edge incident to a 4-vertex, changing it to a 3-vertex and destroying half an edge. The last four rows of the table also capture I-reductions. 0-reductions are irrelevant to the table, which does not consider vertices of degree 0.

The sequence of reductions reducing a graph to a vertexless graph can be parametrized by an 11-vector \vec{n} giving the number of reductions (and partial reductions) indexed by the rows of the table, so for example its first element is the number of III-reductions on 4-vertices whose neighbors are also all 4-vertices. Since the reductions destroy all m edges, the dot product of \vec{n} with the table's column "destroys e " (call it \vec{e}) must be precisely m . Since all vertices of degree 4 are destroyed, the dot product of \vec{n} with the column "destroys 4" (call it \vec{d}_4) must be ≥ 0 , and the same goes for the "destroy" columns 3, 2 and 1. The number of III-reductions is the dot product of \vec{n} with the "steps" column, $\vec{n} \cdot \vec{s}$. How large can the number of III-reductions $\vec{n} \cdot \vec{s}$ possibly be?

To find out, let us maximize $\vec{n} \cdot \vec{s}$ subject to the constraints that $\vec{n} \cdot \vec{e} = m$ and that $\vec{n} \cdot \vec{d}_4$, $\vec{n} \cdot \vec{d}_3$, $\vec{n} \cdot \vec{d}_2$ and $\vec{n} \cdot \vec{d}_1$ are all ≥ 0 . Instead of maximizing over proper reduction collections \vec{n} ,

which seem hard to characterize, we maximize over the larger class of non-negative real vectors \vec{n} , thus obtaining an upper bound on the proper maximum. Maximizing the linear function $\vec{n} \cdot \vec{s}$ of \vec{n} subject to a set of linear constraints (such as $\vec{n} \cdot \vec{e} = m$ and $\vec{n} \cdot \vec{d}_4 \geq 0$) is simply solving a linear program (LP); the LP's constraint matrix and objective function are the parts of Table 2 right of the double line. To avoid dealing with “ m ” in the LP, we set $\vec{n}' = \vec{n}/m$, and solve the LP with constraints $\vec{n}' \cdot \vec{e} = 1$, and as before $\vec{n}' \cdot \vec{d}_4 \geq 0$, etc., to maximize $\vec{n}' \cdot \vec{s}$. The “ \vec{n}' ” LP is a small linear program (11 variables and 5 constraints) and its maximum is precisely $1/5$, showing that the number of III-reduction steps — $\vec{n} \cdot \vec{s} = m\vec{n}' \cdot \vec{s}$ — is at most $m/5$.

That the LP's maximum is at most 5 can be verified from the LP's dual solution of $\vec{y} = (0.20, 0, -0.05, -0.2, -0.1)$. It is easy to check that in each row, the “steps” value is less than or equal to the dot product of this dual vector with the “destroys” values. That is, writing D for the whole “destroys” constraint matrix, we have $\vec{s} \leq D\vec{y}$. Thus, $\vec{n}' \cdot \vec{s} \leq \vec{n}' \cdot (D\vec{y}) = (\vec{n}'D) \cdot \vec{y}$. But $\vec{n}'D$ must satisfy the LP's constraints: its first element must be 1 and the remaining elements non-negative. Meanwhile, the first element of \vec{y} is 0.2 and its remaining elements are non-positive, so $\vec{n}' \cdot \vec{s} \leq (\vec{n}'D) \cdot \vec{y} \leq 0.2$. This establishes that the number of type-III reductions can be at most $1/5$ th the number of edges m , concluding the proof. \square

Theorem 5. *A Max 2-CSP instance on n variables with m dyadic constraints and length L can be solved in time $O(nr^{3+m/5})$ and space $O(L)$.*

Proof. The theorem is an immediate consequence of Claim 3 and Lemma 4. \square

The LP's *dual* solution gives a “potential function” proof of Lemma 4. The dual assigns “potentials” to the graph's edges and to vertices according to their degrees, such that the number of steps counted for a reduction is at most its change to the potential. Since the potential is initially at most $0.20m$ and finally 0 , the number of steps is at most $m/5$. (Another illustration of duality appears in the proof of Lemma 20.)

The *primal* solution of the LP, which describes the worst case, uses (proportionally) 1 III-reduction on a 4 -vertex with all 4 -neighbors, 1 III-reduction on a 3 -vertex, and 3 II-reductions (the actual values are $1/10$ th of these). As it happens, this LP worst-case bound is achieved by the complete graph K_5 , whose 10 edges are destroyed by two III-reductions and then some I- and II-reductions.

5. AN $O(nr^{5+19m/100})$ ALGORITHM

5.1. Improving Algorithm A. The analysis of Algorithm A contains the seeds of its improvement. First, since reduction on a 5 -vertex may destroy only 5 edges, we can no longer ignore such reductions if we want to improve on $m/5$. This simply means including them in the LP.

Second, were this the only change we made, we would find the LP solution to be the same as before (adding new rows leaves the previous primal solution feasible). The solution is supported on a “bad” reduction destroying only 4 edges (reducing on a 4 -vertex with all 4 -neighbors), while the other reductions it uses are more efficient. This suggests that we should focus on eliminating the bad reduction. Indeed, if in the LP we ascribe 0 “steps” to the bad reduction instead of 1 , the LP cost decreases to $23/120$ (about 0.192), and support of the new solution includes reductions on a degree- 5 vertex with all degree- 5 neighbors and on a degree- 4 vertex with one degree- 3 neighbor, each resulting in the destruction of only 5 edges. Counting 0 steps instead of 1 for this degree- 5 reduction gives the LP a cost of $19/100$, suggesting that if we could somehow avoid this reduction too, we might be able to hope for an algorithm running in time $O^*(r^{19m/100})$; in fact our algorithm will achieve this. Further improvements could come from avoiding the next bad cases — a 5 -vertex with neighbors of degree 5 except for one of degree 4 , and a 4 -vertex with neighbors of degree 4 except for one of degree 3 — but we have not pursued this.

Finally, we will also need to take advantage of the component structure of our graph. For example, a collection of many disjoint K_5 graphs requires $m/5$ III-reductions in total. To beat $O^*(r^{m/5})$ we will have to use the fact that an optimum solution to a disconnected CSP is a union of solutions of its components, and thus that the $m/5$ reductions can in some sense be done in parallel, rather than sequentially. Correspondingly, where Algorithm A built a *sequence* of reductions of length at most $m/5$, Algorithm B will build a *reduction tree* whose III-reduction *depth* is at most $2 + 19m/100$. The depth bound is proved by showing that in any sequence of reductions in a component on a fixed vertex, all but at most two “bad” reductions can be paired with other reductions, and for the good reductions (including the paired ones), the LP has maximum $19/100$.

5.2. Algorithm B: General description. Like Algorithm A, Algorithm B preferentially performs type 0, I or II reductions, but it is more particular about the vertices on which it III-reduces. When forced to perform a type III reduction, Algorithm B selects a vertex in the following decreasing order of preference:

- a vertex of degree ≥ 6 ;
- a vertex of degree 5 with at least one neighbor of degree 3 or 4;
- a vertex of degree 5 whose neighbors all have degree 5;
- a vertex of degree 4 with at least one neighbor of degree 3;
- a vertex of degree 4 whose neighbors all have degree 4;
- a vertex of degree 3.

When Algorithm B makes any such reduction with any degree-3 neighbor, it immediately follows up with II-reductions on all those neighbors.¹ Algorithm B then recurses separately on each component of the resulting graph.

As before, in order to get an efficient implementation we must be careful about details. Section 5.3 discusses the construction of the “reduction tree”; a reader only interested in an $O^*(r^{19m/100})$ bound could skip Lemma 6 there. Section 5.4 is essential, and gives the crucial bound $19m/100 + O(1)$ on the depth of a reduction tree, while Section 5.5 establishes that if the depth of a reduction tree is d then an optimal score can be found in time $O^*(r^d)$. Finally, Section 5.6 ties up loose ends (including how to move from an optimal score to an optimal assignment) and gives the main result of this part of the paper (Theorem 11).

5.3. Algorithm B: First phase. As with Algorithm A, a first phase Algorithm B.1 of Algorithm B starts by identifying a sequence of graph reductions. Because Algorithm B will treat graph components individually, Algorithm B.1 then organizes this *sequence* of reductions into a *reduction tree*. The tree has vertices in correspondence with those of G , and the defining property that if reduction on a (graph) vertex v divides the graph into k components, then the corresponding tree vertex v has k children, one for each component, where each child node corresponds to the first vertex reduced upon in its component (*i.e.* the first vertex in the reduction sequence restricted to the set of vertices in the component). If the graph is initially disconnected, the reduction “tree” is really a forest, but since this case presents no additional issues we will speak in terms of a tree. We remark that the number of child components k is necessarily 1 for I- and II-reductions, can be 1 or more for a III-reduction, and is 0 for a 0-reduction.

We define the *III-reduction depth* of an instance to be the maximum number of III-reduction nodes in any root-to-leaf path in the reduction tree. Lemma 6 characterizes an efficient construction of the tree, but it is clear that it can be done in polynomial time and space. The crux of the matter is Lemma 7, which relies on the reduction preference order set forth above, but not on the algorithmic details of Algorithm B.1.

¹An example of this was shown in Figure 1. In some cases, we may have to use I-reductions or 0-reductions instead of II-reductions (for instance if the degree-3 neighbors contain a cycle), but the effect is still to destroy one edge and one vertex for each degree-3 neighbor.

Algorithm B.1: Algorithm B, first phase

- 1: **Input** a constraint graph $G_0 = G$.
 - 2: **for** $i = 1$ to n **do**
 - 3: **Select** a vertex v_i by the preference order described above.
 - 4: **Reduce** G_{i-1} on v_i to produce G_i .
 - 5: **end for**
 - 6: **Initialize** T to be an empty forest.
 - 7: **for** $i = n$ to 1 **do**
 - 8: **Reverse** the i th reduction. For a 0-reduction on v_i , add an isolated node v_i to the forest T . For a I-reduction on v_i with neighbor x_i in G_{i-1} , set v_i to be the parent of the root node of the component of T containing x_i . Do the same for a II-reduction on v_i , whose G_{i-1} -neighbors x_i and y_i will belong to a common component of T . For a III-reduction on v_i , unite all component trees of T containing G_{i-1} -neighbors of v_i by setting v_i as the common parent of their roots. (Details of an efficient implementation are in the proof of Lemma 6.)
 - 9: **end for**
 - 10: **Output** the reduction tree T . T has the property that for any node $v \in T$, reducing G on all ancestor nodes of the corresponding node $v \in G$ produces a graph G' whose component containing v has vertex set equal to the vertex set of the subtree of T rooted at v .
-

Lemma 6. *A reduction tree on n vertices which has III-reduction depth d can be constructed in time $O(dn + n)$ and space $O(m + n)$.*

Proof. We use Algorithm B.1 (see displayed pseudocode). First the sequence of reductions is found much as in Algorithm A.1 and in the same time and space (see Claim 1). As long as there are any vertices of degree ≥ 6 this works exactly as in Algorithm A.1, but with stacks up to degree 6. Once the degree-6 stack is empty it will remain empty (no reduction increases any vertex degree) and at this point we create stacks according to the degree of a vertex and the degrees of its neighbors (for example, a stack for vertices of degree 5 with two neighbors of degree 5 and one neighbor each with degrees 4, 3 and 2). Since the degrees are bounded by 5 this is a small constant number of stacks, which can be initialized in linear time. After that, for each vertex whose degree is affected by a reduction (and which thus required processing time $\Omega(1)$ in Algorithm A.1), we must update the stacks for its at most 5 neighbors (time $O(1)$); this does not change the complexity.

To form the reduction tree we read backwards through the sequence of reductions growing a collection of subtrees, starting from the leaves, gluing trees together into larger ones when appropriate, and ending with the final reduction tree. We now describe this in detail, and analyze the time and space of Algorithm B.1.

Remember that there is a direct correspondence between reductions, vertices of the CSP's constraint graph, and nodes in the reduction tree. At each stage of the algorithm we have a set of subtrees of the reduction tree, each subtree labeled by some vertex it contains. We also maintain a list which indicates, for each vertex, the label of the subtree to which it belongs, or "none" if the corresponding reduction has not been reached yet. Finally, for each label, there is a pointer to the corresponding tree's root.

Reading backwards through the sequence of reductions, we consider each type of reduction in turn.

0-reduction: A forward 0-reduction on y destroys the isolated vertex y , so the reverse reduction creates a component consisting only of y . We create a new subtree consisting only of y , label it " y ", root it at y , and record that y belongs to that subtree.

I-reduction: If we come to a I-reduction on vertex y with neighbor x , note that x must already have been seen in our backwards reading and, since I-reductions do not divide components, the reversed I-reduction does not unite components. In this case we identify

the tree to which x belongs, leave its label unchanged, make y its new root, make the previous root v (typically $v \neq x$) the sole child of y , update the label root-pointer from v to y , and record that y belongs to this tree.

II-reduction: For a II-reduction on vertex y with neighbors x and z , the forward reduction merely replaces the x - y - z path with the edge x - y and thus does not divide components. Thus the reversed reduction does not unite components, and so in the backwards reading x and z must already belong to a common tree. We identify that tree, leave its label unchanged, make y its new root, make the previous root the sole child of y , and record that y belongs to this tree.

III-reduction: Finally, given a III-reduction on vertex y , we consider y 's neighbors x_i , which must previously have been considered in the backwards reading. We unite the subtrees for the x_i into a single tree with root y , y 's children consisting of the roots for the labels of the x_i . (If some or all the x_i already belong to a common subtree, we take the corresponding root just once. Since the roots are values between 1 and n , getting each root just once can be done without any increase in complexity using a length- n array; this is done just as we eliminated parallel edges on a vertex in Algorithm A's first phase — see the proof of Claim 1.) We give the resulting tree the new label y , abandon the old labels of the united trees, and point the label y to the root y . Relabeling the tree also means conducting a depth-first search to find all the tree's nodes and update the label information for each. If the resulting tree has size n' the entire process takes time $O(n')$.

In the complete reduction tree, define “levels” from the root based only on nodes corresponding to III-reductions (as if contracting out nodes from 0, I and II-reductions). The III-reduction nodes at a given level of the tree have disjoint subtrees, and thus in the “backwards reading” the total time to process all of these nodes together is $O(n)$. Over d levels, this adds up to $O(dn)$. The final time bound $O(dn + n)$ also accommodates time to process all $O(n)$ 0-, I- and II-reductions.

The space requirements are a minimal $O(n)$: beyond the space implicit in the input and that entailed by the analog of Algorithm A.1, the only space needed is the $O(n)$ to maintain the labeled forest. \square

5.4. Reduction-tree depth. Analogous to Lemma 4 characterizing Algorithm A, the next lemma is the heart of the analysis of Algorithm B.

Lemma 7. *For a graph G with m edges, the reduction tree's III-reduction depth is $d \leq 2 + 19m/100$.*

Proof. By the same reasoning as in the proof of Lemma 4, it suffices to prove the lemma for graphs with maximum degree at most 5.

Define a “bad” reduction to be one on a 5-vertex all of whose neighbors are also of degree 5, or on a 4-vertex all of whose neighbors are of degree 4. (These two reductions destroy 5 and 4 edges respectively, while most other reductions, coupled with the II-reductions they enable, destroy at least 6 edges.) The analysis is aimed at controlling the number of bad reductions. In particular, we show that every occurrence of a bad reduction can be paired with one or more “good” reductions, which delete enough edges to compensate for the bad reduction.

For shorthand, we write reductions in terms of the degree of the vertex on which we are reducing followed by the numbers of neighbors of degrees 5, 4, and 3, so for example the bad reduction on a 5-vertex is written (5|500). Within a component, a (5|500) reduction is performed only if there is no 5-vertex adjacent to a 3- or 4-vertex; this means the component *has* no 3- or 4-vertices, since otherwise a path from such a vertex to the 5-vertex would include an edge incident on a 5-vertex and a 3- or 4-vertex.

We bound the depth by tracking the component containing a fixed vertex, say vertex 1, as it is reduced. Of course the same argument (and therefore the same depth bound) applies to every

vertex. If the component necessitates a “bad 5-reduction” (a bad III-reduction on a vertex of degree 5), one of four things must be true:

- C1:** This is the first degree-5 reduction in this branch of the reduction tree.
- C2:** The previous III-reduction (the first III-reduction ancestor in the reduction tree, which because of our preference order must also have been a degree-5 reduction) was on a (5|005) vertex, and left no vertices of degree 3 or 4.
- C3:** The previous III-reduction was on a 5-vertex and produced vertices of degree 3 or 4 in this component, but they were destroyed by I- and II-reductions.
- C4:** The previous III-reduction was on a 5-vertex and produced vertices of degree 3 or 4, but split them all off into other components.

As in the proof of Lemma 4, for each type of reduction we will count: its contribution to the depth (normally 1 or 0, but we also introduce “paired” reductions counting for depth 2); the number of edges it destroys; and the number of vertices of degree 4, 3, 2, and 1 it destroys. Table 3 shows this tabulation. In Algorithm B we immediately follow each III-reduction with a II-reduction on each 2-vertex it produces, so for example in row 1 a (5|005) reduction destroys a total of 10 edges and 5 3-vertices; it also momentarily creates 5 2-vertices but immediately reduces them away.

The table’s boldfaced rows and the new column “forces” require explanation. They relate to the elimination of the bad (5|500) reduction from the table, and its replacement with versions corresponding to the cases above.

Case (C1) above can occur only once. Weakening this constraint, we will allow it to occur any number of times, but we will count its depth contribution as 0, and add 1 to the depth at the end. For this reason, the first bold row in Table 3 has depth 0 not 1.

In case (C2) we may pair the bad (5|500) reduction with its preceding (5|005) reduction. This defines a new “pair” reduction shown as the second bold row of the table: it counts for 2 steps, destroys 15 edges, etc. (Other, non-paired (5|005) good reductions are still allowed as before.)

In case (C3) we wish to similarly pair the (5|500) reduction with a I- or II-reduction, but we cannot say specifically with which sort. The “forces” column of Table 3 will constrain each (5|500) reduction for this case to be accompanied by at least one I-reduction (two half-edge reductions of any sort) or II-reduction.

In case (C4), the (5|500) reduction produces a non-empty side component destroyed with the usual reductions but adding depth 0 to the component of interest. These reductions can be expressed as a nonnegative combination of half-edge reductions, which must destroy at least one edge, so we force the (5|500) reduction to be accompanied by at least two half-edge reductions, precisely as in case (C3). Thus case (C4) does not require any further changes to the table.

Together, the four cases mean that we were able to exclude (5|500) reductions, replacing them with less harmful possibilities represented by the first three bold rows in the table.

We may reason identically for bad (4|040) reductions on 4-vertices, contributing the other three bold rows. We reiterate the observation that I-reductions, as well as the merging of parallel edges, can be written as a nonnegative combination of half-edge reductions.

In analyzing a leaf of the reduction tree, let vector \vec{n} count the number of reductions of each type, as in the proof of Lemma 4. As before, the dot product of \vec{n} with the “destroys e ” column is constrained to be 1 (we will skip the version where it is m and go straight to the normalized form), its dot products with the other “destroys” columns must be non-negative, ditto its dot product with the “forces” column, and the question is how large its dot product x with the “depth” column can possibly be. For then, unnormalizing, the splitting-tree depth of vertex 1 as we counted it is at most xm , and the true III-reduction depth (accounting for the possible case (C1) occurrences for 4- and 5-vertices) is at most $2 + xm$.

line #	deg	#nbrs of deg					destroys					forces	depth
		5	4	3	2	1	e	4	3	2	1		
1	5	0	0	5	0	0	10	0	5	0	0	0	1
2	5	0	1	4	0	0	9	1	3	0	0	0	1
3	5	0	2	3	0	0	8	2	1	0	0	0	1
4	5	0	3	2	0	0	7	3	-1	0	0	0	1
5	5	0	4	1	0	0	6	4	-3	0	0	0	1
6	5	0	5	0	0	0	5	5	-5	0	0	0	1
7	5	1	0	4	0	0	9	-1	4	0	0	0	1
8	5	1	1	3	0	0	8	0	2	0	0	0	1
9	5	1	2	2	0	0	7	1	0	0	0	0	1
10	5	1	3	1	0	0	6	2	-2	0	0	0	1
11	5	1	4	0	0	0	5	3	-4	0	0	0	1
12	5	2	0	3	0	0	8	-2	3	0	0	0	1
13	5	2	1	2	0	0	7	-1	1	0	0	0	1
14	5	2	2	1	0	0	6	0	-1	0	0	0	1
15	5	2	3	0	0	0	5	1	-3	0	0	0	1
16	5	3	0	2	0	0	7	-3	2	0	0	0	1
17	5	3	1	1	0	0	6	-2	0	0	0	0	1
18	5	3	2	0	0	0	5	-1	-2	0	0	0	1
19	5	4	0	1	0	0	6	-4	1	0	0	0	1
20	5	4	1	0	0	0	5	-3	-1	0	0	0	1
21	5	5	0	0	0	0	5	-5	0	0	0	0	0
22	5 + 5	5	0	5	0	0	15	-5	5	0	0	0	2
23	5	5	0	0	0	0	5	-5	0	0	0	-1	1
24	4	0	0	4	0	0	8	1	4	0	0	0	1
25	4	0	1	3	0	0	7	2	2	0	0	0	1
26	4	0	2	2	0	0	6	3	0	0	0	0	1
27	4	0	3	1	0	0	5	4	-2	0	0	0	1
28	4	0	4	0	0	0	4	5	-4	0	0	0	0
29	4 + 4	0	4	4	0	0	12	6	0	0	0	0	2
30	4	0	4	0	0	0	4	5	-4	0	0	-1	1
31	3	0	0	3	0	0	6	0	4	0	0	0	1
32	2	0	0	0	0	0	1	0	0	1	0	1	0
33	$\frac{1}{2}e$	1	0	0	0	0	$\frac{1}{2}$	-1	0	0	0	$\frac{1}{2}$	0
34	$\frac{1}{2}e$	0	1	0	0	0	$\frac{1}{2}$	1	-1	0	0	$\frac{1}{2}$	0
35	$\frac{1}{2}e$	0	0	1	0	0	$\frac{1}{2}$	0	1	-1	0	$\frac{1}{2}$	0
36	$\frac{1}{2}e$	0	0	0	1	0	$\frac{1}{2}$	0	0	1	-1	$\frac{1}{2}$	0
37	$\frac{1}{2}e$	0	0	0	0	1	$\frac{1}{2}$	0	0	0	1	$\frac{1}{2}$	0

TABLE 3. Tabulation of the effects of various reductions in Algorithm B.

As before, x is found by solving the LP: it is 19/100. The dual solution, with weights (0.190, -0.005, -0.035, 0, 0, 0.150) on edges, degrees 4, 3, 2, 1, and “forces”, witnesses this as the maximum possible. (For more on duality, see the proof of Lemma 20.) This concludes the proof. \square

We observe that the maximum is achieved by a weight vector with just three nonzero elements, putting relative weights of 8, 6, and 5 on the reductions (5|410), (4|031), and (3|003). That is, the proof worked by essentially eliminating bad reductions of types (5|500) and (4|040) (which destroy only 5 and 4 edges respectively, in conjunction with the II-reductions they enable), and the bound produced uses the second-worst reductions, of types (5|410) and (4|031) (each destroying

5 edges, with the accompanying II-reductions), which it is forced to balance out with favorable III-reductions of type (3|003).

Remark 8. For an m -edge graph G and maximum degree ≤ 4 , the reduction tree's III-reduction depth is $d \leq 1 + (3/16)m$. If G has maximum degree ≤ 3 , the depth is $d \leq m/6$.

Proof. The first statement's proof is identical to that of Lemma 7 except that from Table 3 we discard reductions (rows) involving vertices of degree 5, we solve the new LP, and we have an additive 1 instead of 2 (for a single bad reduction on a vertex of degree 4, rather than one each for degrees 4 and 5). The second statement can be obtained directly and trivially, or we may go through the same process. \square

5.5. Algorithm B: Second phase. It is straightforward to compute the optimal *score* of an instance; this is Algorithm B.2 (see displayed pseudocode). As with Algorithm A.2, Algorithm B.2

Algorithm B.2: Algorithm B, second phase

```

1: Input: The input consists of a CSP instance  $(G, S)$ , a tree  $T$ , and a vertex  $v \in T$  such that
   the subtree of  $T$  rooted at  $v$  is a reduction tree for the component of  $(G, S)$  containing  $v$ . (We
   start with an initial CSP  $(G_0, S_0)$  with reduction tree  $T$ , and  $(G, S)$  is the reduction of  $(G_0, S_0)$ 
   on the ancestors of  $v$ , with some choices of colors for the III-reductions.)
2: Let  $v'$  be the first 0- or III-reduction node below (or equal to)  $v$ .
3: I- and II-reduce on all nodes from  $v$  up to but not including  $v'$ . (If  $v = v'$ , do nothing.)
4: if  $v'$  is a 0-reduction node then
5:   Reduce on  $v'$  and return the resulting niladic score  $s$ .
6: end if
7: Let  $v_1, \dots, v_k$  be the children of  $v'$ . Let  $s := -\infty$ .
8: for color  $C = 1$  to  $r$  do
9:   III-reduce on  $v'$  with color  $C$ .
10:  Let  $s' := 0$ .
11:  for  $i = 1, \dots, k$  do
12:    Let  $s' := s' + \text{B.2}(v_i)$ , computed recursively.
13:  end for
14:  Let  $s := \max\{s, s'\}$ ,
15: end for
16: Output:  $s$ , the optimal score of the component of  $G$  containing  $v$ .
```

is a recursive procedure which, with the exception of a minimal amount of state information, works “in place” in the global data structure for the problem instance. In addition to the algorithm's explicit input, state information is a single active node $v^* \in T$ (a descendant of v), and, for each ancestor of v^* : a reference to which of its children leads to v^* ; the sum of the optimal scores for the earlier children; its current color; and the usual information needed to reverse the reduction.

The recursion can be executed with a global state consisting of a path from the root node to the currently active node, along with a color for each III-reduction node along the path: after the current node v^* and color have been explored, if possible the color is incremented, otherwise if there is a next sibling of v^* it is tried with color 1, otherwise control passes to the first III-reduction ancestor of v' , and if there is no such ancestor then the recursion is complete.

Define the depth d of a tree node v to be the maximum, over all leaves ℓ under v , of the number of III-reduction nodes from v to ℓ inclusive. The following claim governs the running time of Algorithm B.2.

Claim 9. For a tree node v of depth d whose subtree has order n_v , Algorithm B.2 runs in time $O(n_v r^{3+d})$ and in linear space.

Proof. Any sequence of 0-, I- and II-reductions can be performed in time $O(r^3n)$, and a set of r III-reductions (one for each color) in time $O(r^2n)$ (see the proof of Claim 2). Let us “renormalize” time so that the sum of these two can be bounded simply by r^3n (again as in the proof of Claim 2). We will prove by induction on d that an instance of order n_v and depth d can be solved in time at most

$$f(n_v, d) := r^3n_v(r^d + (r^{d+1} - r)/(r - 1)), \quad (3)$$

which is at most $3n_v r^{3+d}$.

The base case is that $d = 0$, no III-reductions are required, and the instance is solved by performing and reversing a series of 0-, I- and II-reductions; this takes time $\leq r^3n_v$, which is smaller than the right-hand side of (3).

For a node v of depth $d > 0$, define v' to be the first III-reduction descendant of v (or v itself if v is a III-reduction node). The reductions from v up to but not including v' , and the r possible reductions on v' , take time $\leq r^3n$. The total time taken by Algorithm B.2 is this plus the time to recursively solve each of the r subinstances reduced from v' . If the tree node v' has outdegree k , each of the r subinstances decomposes into k components, the i th component having order n_i and depth d_i (with $n_1 + \dots + n_k = n_v - 1$, and $d_i \leq d - 1$), and thus the total time taken is $f(n_v, d) \leq r^3n + r \sum_{i=1}^k f(n_i, d_i)$. By the inductive hypothesis (3), then,

$$\begin{aligned} f(n_v, d) &\leq r^3n_v + r \sum_{i=1}^k f(n_i, d_i) \\ &\leq r^3n_v + r \sum_{i=1}^k r^3n_i(r^{d_i} + (r^{d_i+1} - r)/(r - 1)) \\ &\leq r^3n_v + r \sum_{i=1}^k r^3n_i(r^{d-1} + (r^d - r)/(r - 1)) \\ &< r^3n_v + (r^3n_v)r(r^{d-1} + (r^d - r)/(r - 1)) \\ &\leq r^3n_v(1 + r^d + (r^{d+1} - r^2)/(r - 1)) \\ &\leq r^3n_v(r^d + (r^{d+1} - r)/(r - 1)). \end{aligned}$$

The linear space demand follows just as for Algorithm A.2. □

5.6. Algorithm B: Third phase. In Algorithm A, the moment an optimal score is achieved (at the point of reduction to an empty instance), all III-reduction vertices already have their optimal colors, and reversing all reductions gives an optimal coloring. This approach does not work for Algorithm B, because we now have a tree of reductions rather than a path of reductions.

Imagine, for example, 3-coloring a III-reduction vertex A with children B and C that are also III-reduction vertices, and where the optimal colors are $\phi(A) = 1$, $\phi(B) = 2$, $\phi(C) = 3$. We first try the coloring $\phi(A) = 1$, and within this we try the six (not nine!) combinations $\phi(B) = 1, 2, 3$ and then $\phi(C) = 1, 2, 3$. Even knowing the optimal score, there is no “moment of truth” when the score is achieved: we have gone past $\phi(B) = 2$ by the time we start with $\phi(C) = 1$. Also, even if we could recover the fact that for $\phi(A) = 1$ the optimal settings were $\phi(B) = 2$, $\phi(C) = 3$, we would not be able to remember this as we were trying $\phi(A) = 2, 3$. (In this simple example we would already be forced to remember optimal choices for both B and C for each possible color of A , and taking the full tree into account this would become an exponential memory requirement.)

Fortunately, there is a relatively simple work-around. Having computed the optimal score with Algorithm B.2, we can try different colors at the highest III-reduction vertex to see which gives that score; this gives the optimal coloring of that vertex. (It is worth noting that we cannot

Algorithm B.3: Algorithm B, third phase

```

1: Input a CSP  $(G, S)$  and a reduction tree  $T$  for  $G$ .
2: for each III-reduction node  $v \in T$ , in depth-first search order (by first visit) do
3:   Let  $s := \text{B.2}(v)$ .
4:   Let  $v_1, \dots, v_k$  be the children of  $v$ .
5:   for color  $C = 1$  to  $r$  do
6:     III-reduce on  $v$  with color  $C$ .
7:     if  $s = \text{B.2}(v_1) + \dots + \text{B.2}(v_k)$  then
8:       Assign color  $C$  to  $v$  and break.
9:     end if
10:  end for
11: end for
12: At this point all III-reduction nodes of  $G$  are colored, optimally.
13: Perform all corresponding III-reductions on  $G$ , using these optimal colors, to derive an equivalent instance  $G'$ .
14: Perform the 0-, I- and II-reductions of  $T$ , in depth-first search order, reducing  $G'$  to an empty instance.
15: Reverse the 0-, I- and II-reductions to optimally color all vertices of  $G'$ , and thus of  $G$ .
16: Output the coloring of  $G$ .

```

immediately reverse the ancestor I- and II-reductions, as those vertices may be adjacent to vertices not yet colored; coloring by reversing reductions only works after we have reduced to an empty instance.) We can repeat this procedure, working top down, to optimally color all III-reduction vertices. After this, it is trivial to color all the remaining, 0-, I- and II-reduction vertices. These stages are all described as Algorithm B.3 (see displayed pseudocode).

Correctness of this recursive algorithm is immediate from the score-preserving nature of the reductions.

Claim 10. *For a CSP instance (G, S) where G has n nodes and m edges, and whose reduction tree per Algorithm B.1 has depth d , Algorithm B.3 runs in time $O(nr^{3+d})$ and in linear space, $O(L)$.*

Our main result follows immediately from Lemma 6, Lemma 7 (or Remark 8 for graphs with maximum degree 4 or less), and Claims 9 and 10.

Theorem 11. *Algorithm B solves a Max 2-CSP instance (G, S) , where G has n vertices and m edges, in time $O(nr^{5+19m/100})$ and in linear space, $O(L)$. If G has maximum degree 4 the time bound may be replaced by $O(nr^{4+3m/16})$, and if G has maximum degree 3, by $O(nr^{3+m/6})$.*

6. VERTEX-PARAMETRIZED RUN TIME

In most of this paper we consider run-time bounds as a function of the number of edges in a Max 2-CSP instance's constraint graph, but we briefly present a couple of results giving time bounds as a function of the number of *vertices*, along with the average degree d and (for comparison with existing results) the maximum degree Δ .

For general Max 2-CSPs, we derive a run-time bound by using the following lemma in lieu of Lemma 7. (Thus, the linear-programming analysis plays no role here; we are simply using the power of our reductions. Because the lemma bounds the *number* of III-reductions, not just their depth, it will also suffice to use Algorithm A instead of the more complicated Algorithm B.)

Lemma 12. *For a graph G of order n , with average degree $d \geq 2$, in time $\text{poly}(n)$ we can find a reduction sequence with at most $(1 - \frac{2}{d+1})n$ III-reductions.*

Proof. Let $\alpha_2(G)$ be the maximum number of vertices in an induced forest in G . This quantity was investigated by Alon, Kahn and Seymour [AKS87], who showed that

$$\alpha_2(G) \geq \sum_{v \in V(G)} \min \left\{ 1, \frac{2}{d(v) + 1} \right\},$$

and that there is a polynomial-time algorithm for finding an induced forest of the latter size (in fact, they proved a rather more general result; this is the special case of their Theorem 1.3 with degeneracy parameter 2). It follows (same special case of their Corollary 1.4) that if G has average degree $d \geq 2$ then

$$\alpha_2(G) \geq \frac{2n}{d+1}.$$

Note that this is sharp when G is a union of complete graphs of order $d+1$.

Now we simply III-reduce on every vertex of G *not* in the induced subgraph (or 0-, I- or II-reduce on such a vertex which has degree < 3 by the time we reduce on it). After this sequence of reductions, the graph is a forest, and 0-, I- and II-reductions suffice to reduce it to the empty graph. Thus the total number of III-reductions needed is $\leq n - \alpha_2(G) \leq n(1 - \frac{2}{d+1})$. \square

Theorem 13. *A Max 2-CSP instance with constraint graph G of order n with average degree $d \geq 2$ can be solved in time*

$$O \left(nr^{(1-\frac{2}{d+1})n} + \text{poly}(n) \right).$$

Proof. Immediate from Lemma 12 and Claim 3. (Since Lemma 12 gives a bound on the number of III-reductions, not merely the depth, it suffices to use Algorithm A rather than the more complicated Algorithm B.) \square

Note that for $d < (\sqrt{17561} + 181)/38 \approx 8.25$, Theorem 11 gives a smaller bound than Theorem 13, while for $d < 100/31 \approx 3.23$ the best bound is given by our $O^*(r^{(d-2)n/4})$ algorithm from [SS03, SS06c] (there stated more precisely as $O(nr^{(m-n)/2})$).

Theorem 13 improves upon one recent result of Della Croce, Kaminski, and Paschos [DCKP], which solves Max Cut (specifically) in time $O^*(2^{mn/(m+n)}) = O^*(2^{(1-\frac{2}{d+2})n})$. A second algorithm from [DCKP] solves Max Cut in time $O^*(2^{(1-2/\Delta)n})$, where Δ is the constraint graph's maximum degree; this is better than our general algorithm if the constraint graph is “nearly regular”, with $\Delta < d+1$.

Our results also improve upon a recent result of Fürer and Kasiviswanathan [FK07], which, for binary Max 2-CSPs, claims a running time of $O^*(2^{(1-\frac{1}{d-1})n})$ (when $d > 2$ and the constraint graph is connected, per personal communication). For $d > 3$ the bound of Theorem 13 is smaller, while for $2 < d \leq 3$ (in fact, for d up to 5), our $O^*(2^{n(d-2)/4})$ algorithm from [SS03, SS06c] is best.

It is also possible to modify the algorithm described by Theorem 11 to give reasonably good vertex-parametrized algorithms for special cases, such as Maximum Independent Set. As remarked in the Introduction, however, there are faster algorithms for MIS.

Corollary 14. *An instance of weighted Maximum Independent Set on an n -vertex graph can be solved in time $O(n2^{3n/8})$ and in linear space, $O(m+n)$.*

Proof. If $n < 20$ we may solve the instance in time $O(1)$, and if the graph's maximum degree is $\Delta \leq 4$ we apply Algorithm B, use Theorem 11's time bound of $O(nr^{4+3m/16})$, and observe that this is $O(n2^{3n/8})$. Otherwise we use a very standard MIS reduction: for any vertex v , either v is not included in the independent set or else it is and thus none of its neighbors is; therefore the maximum weight of an independent set of G satisfies $s(G) = \max\{s(G-v), w(v) + s(G-v-\Gamma(v))\}$, where $w(v)$ is the weight of vertex v and $\Gamma(v)$ is its neighborhood. “Rescaling” time as usual so that we may drop the $O(\cdot)$ notation, if there is a vertex of degree 5 or more, the running time $f(n)$

satisfies $f(n) \leq n + f(n-1) + f(n-6)$. (A relevant constant for this recursion is α : $1 = \alpha^{-1} + \alpha^{-6}$; its value is about 1.285, and in particular less than $2^{3/8}$.) For $n \geq 20$, induction on n confirms that $f(n) \leq n2^{3n/8}$. \square

7. TREewidth AND CUBIC GRAPHS

In this section we show several connections between our LP method, algorithms, and the treewidth of graphs, especially cubic (3-regular) graphs. We first define treewidth, and in Section 7.1 show that it can be bounded in terms of our III-reduction depth. In Section 7.2 we show how a bound on the treewidth of cubic graphs can be incorporated into our LP method to give a treewidth bound for general graphs, and in turn faster (but exponential space) algorithms. In Section 7.3 we show how fast algorithms for cubic graphs generally imply fast algorithms for general graphs, independent of treewidth.

First, we recall the definition of treewidth and introduce the notation we will use. Where $G = (V, E)$, a *tree decomposition* of G is a pair (X, T) , where

- (1) $X = \{X_1, \dots, X_q\}$ is a collection of vertex subsets, called “bags”, covering V , *i.e.*, $X_i \subset V$ and $\bigcup_{i=1}^q X_i = V$;
- (2) each edge of G lies in some bag, *i.e.*, $(\forall uv \in E)(\exists i): \{u, v\} \subset X_i$; and
- (3) T is a tree on vertex set X with the property that if X_j lies on the path between X_i and X_k , then $X_j \supset (X_i \cap X_k)$.

The width of the decomposition tree is defined as $\max_i |X_i| - 1$, and a graph’s treewidth is the minimum width over all tree decompositions. Trees with at least one edge have treewidth 1, and series-parallel graphs have treewidth at most 2.

From Claim 10 we have the following corollary.

Corollary 15. *A CSP whose constraint graph G is a tree or series-parallel graph can be solved in time $O(r^3 n)$ and in linear space.*

Proof. A tree G can be reduced to a vertexless graph by 0- and I-reductions alone: it has III-reduction depth 0. By definition, a series-parallel graph G arises from repeated subdivision and duplication of a single edge. It follows that II-reductions (with their fusings of multiple edges) suffice to reduce G to a collection of isolated edges (disjoint K_2 ’s), which are reduced to the vertexless graph by 0- and I-reductions. Again, G has III-reduction depth 0. \square

7.1. Implications of our results for treewidth. Although trees and series-parallel graphs are both classes of graphs with small treewidth and III-reduction depth 0, there is no reason to think that our algorithm will produce shallow III-reduction depth for all graphs of small treewidth. However, there is an implication in the opposite direction, per Claim 17.

Lemma 16. *If G is 0-reduced to a vertexless graph, $\text{tw}(G) = 0$. If G is I-reduced to G' , $\text{tw}(G) = \max\{1, \text{tw}(G')\}$. If G is II-reduced to G' , $\text{tw}(G) = \text{tw}(G')$. If G is III-reduced to components G_1, \dots, G_s , $\text{tw}(G) \leq 1 + \max_i \text{tw}(G_i)$.*

Proof. For a 0-reduction, G is a single vertex, which has treewidth 0. Otherwise, first note that $\text{tw}(G) \geq \text{tw}(G')$, as shown by the tree decomposition for G' induced by any tree decomposition of G .

For a I-reduction, G adds a pendant edge uv to some vertex v of G' . For any tree decomposition (X', T') of G' , we can form a tree decomposition of G by adding a new bag $X_0 = \{u, v\}$ and linking it to any bag $X'_i \ni v$. This satisfies the defining properties of a tree decomposition, and has treewidth $\max\{1, \text{tw}(G')\}$.

For a II-reduction, G subdivides some edge uv of G' with a new vertex w . We mirror this in the decomposition tree in a way depending on two cases. Either way, (X', T') has a bag containing $\{u, v\}$. If there is any bag of size 3 or more, we simply add a new bag $X_0 = \{u, v, w\}$ and link it to

any bag $X'_i \supseteq \{u, v\}$. If the maximum bag size is 2 then without loss of generality there is a single bag $X'_i = \{u, v\}$, each of whose neighbors may contain either u or v but not both. We replace X'_i with a pair of bags $X_u = \{u, w\}$ and $X_v = \{v, w\}$, join them with an edge, and join the former neighbors of X'_i to either X_u or X_v depending on whether the neighbor contained u or v (if neither, the choice is arbitrary). In either case this shows that $\text{tw}(G) \leq \text{tw}(G')$.

For a III-reduction on a vertex v , let $(X^{(i)}, T^{(i)})$ be tree decompositions of the components G_i resulting from v 's deletion. To obtain a tree decomposition of G , first add v to every bag of every tree; every edge (v, x) can be put in some such bag. Also, create a new bag containing only the vertex v , and join it to one (arbitrarily chosen) bag from each $(X^{(i)}, T^{(i)})$, thus creating a single tree and having the third defining property of a tree decomposition. This shows that $\text{tw}(G) \leq 1 + \max_i \text{tw}(G_i)$. \square

Claim 17. *If a graph G has a reduction tree of III-reduction depth d , then G has treewidth $\leq d + 1$.*

Proof. From the preceding lemma, the treewidth of G is bounded by applying the various treewidth-reduction rules along some critical (though typically not unique) root-to-leaf path in the reduction tree. Traversing that path from leaf to root, the case where treewidth changes from 0 to 1 (from a I-reduction) occurs at most once, and otherwise the treewidth increases only at III-reduction nodes. Thus, $\text{tw}(G)$ is at most 1 plus the maximum, over all root-to-leaf paths, of the number of III-reductions in the path, which is to say $d + 1$. \square

Corollary 18. *A graph G with m edges has treewidth at most $3 + 19m/100$, and a tree decomposition of this width can be produced in time $O(mn + n)$.*

Proof. Immediate from the depth bound of Lemma 7, Algorithm B.1's running time per Lemma 6, and the algorithm in the proof of Claim 17. \square

7.2. Implications from treewidth of cubic graphs. In this section we explore how treewidth bounds for cubic graphs imply treewidth bounds for general graphs. Algorithmic implications of these treewidth bounds are discussed in the next subsection.

Building on a theorem of Monien and Preiss that any cubic (3-regular) graph with m edges has bisection width at most $(1/9 + o(1))m$ [MP06], Fomin and Høie show that such a graph also has pathwidth at most $(1/9 + o(1))m$ [FH06]. (The $o(1)$ terms here are as $m \rightarrow \infty$.) For large m this is significantly better than the treewidth bound of $1 + m/6$ that would result from Claim 17 and the cubic III-reduction depth bound of $m/6$ (each III-reduction on a vertex of degree 3 destroying 6 edges). Since we perform degree-3 III-reductions in a component only when it has no vertices of higher degree, it is possible to use this more efficient treatment of cubic graphs in place of our degree-3 III-reductions, as we now explain.

The result from [FH06] that a 3-regular graph with m edges has pathwidth at most $(1/9 + o(1))m$ implies the following lemma. Since [FH06] relies on a polynomial-time construction, the lemma is also constructive.

Lemma 19. *If every 3-regular graph G with m edges has treewidth at most αm , then any graph G with m edges has treewidth $\text{tw}(G) \leq 3 + \beta(\alpha)m$, and any graph of maximum degree $\Delta(G) \leq 4$ has $\text{tw}(G) \leq 2 + \beta_4(\alpha)m$, where $\beta(\alpha)$ and $\beta_4(\alpha)$ are given by Lemma 20.*

Proof. Recall that our graph reduction algorithm performed III-reductions on vertices of degree 5 and 4 in preference to vertices of degree 3. Build the reduction tree as usual, but terminating at any node corresponding to a graph which is either vertexless or 3-regular. By Lemma 16 and observations in the proof of Lemma 17, the treewidth of the root (the original graph G) is at most 1 plus the maximum, over all root-to-leaf paths, of the “step count” (or “depth”) of each reduction (1 for III-reductions, 0 for other reductions) plus the treewidth of the leaf. If we add a “reduction” taking an m -edge 3-regular graph to a vertexless graph, and count it as αm steps, then $\text{tw}(G)$ is at most 1 plus the maximum over all root-to-leaf paths of the step counts along the path.

We may bound this value by the same LP approach taken previously. We exclude the old degree-3 III-reduction, characterized by line 31 of Table 3. In its place we introduce a family of reductions: for each number of edges m' in a cubic graph (necessarily a multiple of 3) we have a reduction that counts as $\alpha m'$ steps and destroys all m' edges, all $2/3m'$ degree-3 vertices of the cubic graph, and 0 vertices of degrees 4 and 5. As before, going down a path in the reduction tree, any “bad” reduction (a (4|040) or (5|500) reduction) is either paired with a good one to make a combined reduction, or is counted as 0 steps (in at most 2 instances per path). The total number of reduction steps is thus at most 2 plus the step count of a feasible LP solution. Since a row of an LP may be rescaled without affecting the solution value, we may replace the family of 3-regular reductions with a single reduction that counts as α steps, destroys 1 edge and $2/3$ vertices of degree 3, and 0 vertices of degrees 4 and 5. If this LP has optimal solution βm , then the path has true step count $\leq 2 + \beta m$ and G has treewidth $\text{tw}(G) \leq 3 + \beta m$. The proof is completed by Lemma 20, establishing β as a function of α . \square

Lemma 20. *Let LP be the linear program of Table 3 whose line 31 is replaced as below.*

	deg	#nbrs of deg					destroys					forces	depth
		5	4	3	2	1	e	4	3	2	1		
old	3	0	0	3	0	0	6	0	4	0	0	0	1
new	3	0	0	3	0	0	1	0	2/3	0	0	0	α

Then LP has optimal solution

$$\beta(\alpha) = \begin{cases} 7/50 + (3/10)\alpha & 1/9 \leq \alpha \leq 1/5 \\ 13/75 & 0 \leq \alpha \leq 1/9. \end{cases}$$

The same linear program restricted to the constraints corresponding to reductions on vertices of degree 4 and smaller, call it LP_4 , has optimal solution

$$\beta_4(\alpha) = \begin{cases} 1/8 + (3/8)\alpha & 1/9 \leq \alpha \leq 1/5 \\ 1/6 & 0 \leq \alpha \leq 1/9. \end{cases}$$

Proof. To help give a feeling for the interpretation of our linear-programming analysis, we will first give a very explicit duality-based proof, carrying it through for just one of the lemma’s four cases. We will then show a much simpler proof method and apply it to all the cases.

For the first case, it suffices to produce feasible primal and dual LP solutions with the claimed costs. With $1/9 \leq \alpha \leq 1/5$, the primal solution puts weights exactly 0.30, 0.06, 0.08 respectively on the following rows of LP:

deg	#nbrs of deg					destroys					forces	depth
	5	4	3	2	1	e	4	3	2	1		
3	0	0	3	0	0	1	0	2/3	0	0	0	α
4	0	3	1	0	0	5	4	-2	0	0	0	1
5	4	1	0	0	0	5	-3	-1	0	0	0	1

The solution is feasible because the weighted sum of the rows destroys exactly 1 edge and a nonnegative number (in fact, 0) of vertices of each degree. The value of α does not enter into this at all: α does not appear in the constraints, so the primal solution is *feasible* regardless of α . The primal’s *value* is the dot product of (0.30, 0.06, 0.08) with the “depth” column ($\alpha, 1, 1$), and matches the value of β claimed in the lemma.

The dual solution is $\frac{1}{600}[(84, -18, -126, 0, 0, -60) + \alpha(180, 90, 630, 0, 0, 900)]$. It is dual-feasible because, interpreting these values as weights on (respectively) edges, vertices of degree 4, 3, 2, and 1, and forces, for each row of LP the sum of the weights of edges and vertices destroyed, and forces, is

at least the number of steps counted. (The inequality is tight for the rows displayed above, but one must check it for all rows. For some rows, such as Table 3's line 11, corresponding to reduction on a vertex of degree 5 with four neighbors of degree 4 and one of degree 5, the inequality is violated for $\alpha > 1/5$.) The dual LP value is the dot product of the dual solution with the primal's constraint vector $(1, 0, 0, 0, 0, 0)$ (at least 1 edge, 0 vertices of each degree, and 0 "forces" should be destroyed). Thus the dual value is $1 \times (84 + 180\alpha)/600 = 0.14 + 0.30\alpha$, matching the value specified in the lemma, and thus also matching the primal value and proving the solution's optimality.

A much easier proof comes from exploiting a standard and simple fact from linear-programming sensitivity analysis: Suppose a single vector x^* is an optimal solution to two linear programs with the same constraints but different objective functions, given by vectors c_1 and c_2 respectively. Then x^* is also optimal for any linear program where again the constraints are the same, and the objective function c is any convex combination of c_1 and c_2 .²

Thus, to verify the case we have already done, it suffices to check that a single primal solution x^* is optimal for both $\alpha = 1/9$ and $\alpha = 1/5$. This can easily be done by solving LP for some intermediate value, say $\alpha = 1/7$, and checking that the primal x^* obtained, dotted with the objective vector corresponding to $\alpha = 1/9$, is equal to the solution value of the LP for $\alpha = 1/9$, and performing the same check for $\alpha = 1/5$. (Even easier, but not quite rigorous, is simply to solve the LP for, say, $\alpha = 1/9 + 0.001$ and $\alpha = 1/5 - 0.001$, and verify that the two primal solutions are equal.) The remaining cases are verified identically. \square

Corollary 21. *Any graph G with m edges has $\text{tw}(G) \leq (13/75 + o(1))m$, and if $\Delta(G) \leq 4$ then $\text{tw}(G) \leq (1/6 + o(1))m$.*

Proof. Immediate from Lemmas 19 and 20, and the fact that every cubic graph with m edges has treewidth $\leq (1/9 + o(1))m$ [FH06]. The additive constants can be absorbed into the $o(1)m$. \square

We now discuss algorithmic implications of these treewidth bounds.

7.3. Implications from algorithms for cubic graphs. Efficient algorithms for constraint satisfaction of various sorts, and related problems, on graphs of small treewidth have been studied since at least the mid-1980s, with systematic approaches dating back at least to [DP87, DP89, AP89]. A special issue of Discrete Applied Mathematics was devoted to this and related topics in 1994 [AHe94], and the field remains an extremely active area of research.

It is something of a folk theorem that a Max 2-CSP instance of treewidth k can be solved in time and space $O^*(r^k)$ through dynamic programming. (The need for exponential space is of course a serious practical drawback.) Such a procedure was detailed by Jansen, Karpinski, Lingas and Seidel [JKLS05] for solving maximum bisection, minimum bisection, and maximum clique. Those problems are in fact slightly outside the Max 2-CSP framework defined here, but within a broader framework of "Polynomial CSPs" that we explore in [SS06a, SS07]. In [SS06a, SS07] we show how to use dynamic programming on tree decompositions of width k to solve any Polynomial CSP, including the problems above and any Max 2-CSP, in time and space $O^*(r^k)$.

Direct application of dynamic programming in conjunction with Corollary 21 means that any Max 2-CSP can be solved in time and space $O^*(r^{(13/75+o(1))m})$. However, we can do better.

Similarly to how Lemma 19 showed that a cubic treewidth bound αm implies a general treewidth bound of βm , Theorem 22 shows that an $O^*(r^{\alpha m})$ -time algorithm for cubic instances of Max 2-CSP can be used to construct an $O^*(r^{\beta m})$ -time algorithm for arbitrary instances. The approach gives greater generality, since the algorithm for cubic instances need not have anything to do with treewidth. And when the algorithm for cubic instances is tree decomposition-based dynamic

²Proof of this fact is instant: Optimality of x^* for c_1 means that for any feasible x , $c_1 x \leq c_1 x^*$, and likewise for c_2 . Then for any convex combination $c = pc_1 + qc_2$, $p + q = 1$, $p, q \geq 0$, optimality of x^* for c is proved by the observation that for any feasible x , $cx = (pc_1 + qc_2)x = p(c_1 x) + q(c_2 x) \leq p(c_1 x^*) + q(c_2 x^*) = (pc_1 + qc_2)x^* = cx^*$.

programming, this approach gives greater efficiency: we can match the previous paragraph's time bound, while reducing the space requirement (Corollary 23).

Theorem 22. *Given a value $\alpha > 0$, an integer r , and a function $g(m) = O^*(r^{\alpha m})$, suppose there is an algorithm that, for any m -edge 3-regular graph G , solves any CSP with constraint graph G and domain $[r]$ in time $g(m)$. Then there is an algorithm which solves any CSP with domain $[r]$ and any m -edge constraint graph G in time $O^*(r^{\beta(\alpha)m})$, and in time $O^*(r^{\beta_4(\alpha)m})$ if $\Delta(G) \leq 4$, with $\beta(\alpha)$ and $\beta_4(\alpha)$ given by Lemma 20. If the hypothesized algorithm is guaranteed to solve an instance of input size L using space $O(s(L))$, for some nondecreasing function s , then the algorithm assured by the theorem uses space $O(L + s(L))$.*

Proof. The proof is similar to that of Lemma 19. Introduce a family of “reductions” reducing an m -edge 3-regular graph to a vertexless graph and counting for depth αm . Precisely as in the earlier proof, represent them all in the LP by a single reduction destroying 1 edge, $2/3$ vertices of degree 3, and 0 vertices of degree 4 and 5, and counting as depth α .

Reduce a graph G as far as possible by 0-, I- and II-reductions, and III-reductions on vertices of degree 4 and above. For any tree node, and corresponding reduced constraint graph G' , define the depth of G' to be the maximum, over all its 3-regular leaf instances G_i having m_i edges respectively, of αm_i plus the number of III-reductions to get from G to G_i . From our usual LP setup and Lemma 20, it is immediate that any m' -edge graph G' has depth $\leq 2 + \beta m'$.

It remains only to show that depth $2 + \beta m$ implies running time $O^*(r^{2+\beta m})$, and we will do this inductively. Note that a cubic graph with m edges has $n = 2m/3$ vertices, so the fact that $g(m) = O^*(r^{\beta m})$ implies that there is some polynomial $p(n)$ such that $g(m) \leq p(n)f(n, \beta m)$, where f is the function defined by (3) in the proof of Claim 9. Without loss of generality, assume $p(n) \geq 1$. Note that f is given explicitly, and p depends on the bound g guaranteed by the Theorem's hypothesis, but not on r , G , etc.

Suppose the original instance's constraint graph G has n vertices. We now show inductively that each reduced instance G' with n' vertices and depth d' can be solved in time $p(n)f(n', d')$. (We really do mean $p(n)$, not $p(n')$.) The induction begins at the leaves, and proceeds up the tree. For a leaf G' , which is a 3-regular instance, the property is guaranteed by the theorem's hypothesis, $d' = \beta m'$, and $p(n) \geq 1$. Otherwise, for a node G' we may inductively assume the property holds for its children, in which case the running time for G' is at most

$$\begin{aligned} r^3 n' + r \sum p(n)f(n_i, d_i) &\leq p(n)[r^3 n' + r \sum f(n_i, d-1)] \\ &\leq p(n)f(n', d), \end{aligned}$$

where the second inequality is precisely the calculation performed after (3). Taking $G' = G$ shows that the root node G can be solved in time $\leq p(n)f(n, \beta m) = O^*(r^{\beta m})$.

Except for the calls on the hypothesized algorithm, our overall algorithm uses space $O(L)$, per Theorem 5. Since each cubic subinstance has size at most L , and s is nondecreasing, the total space needed is $O(L + s(L))$. \square

Corollary 23. *A Max 2-CSP instance with domain size r and m dyadic constraints can be solved in time $O^*(r^{(13/75+o(1))m})$, and if $\Delta(G) \leq 4$, time $O^*(r^{(1/6+o(1))m})$, in either case in space $O^*(r^{(1/9+o(1))m})$.*

Proof. With $\alpha = 1/9 + o(1)$, Theorem 22's hypothesized algorithm for m -edge cubic instances is given by dynamic programming on a tree decomposition of treewidth $\leq \alpha m$ (which by [FH06] exists and can be found in polynomial space and time), and runs in space and time $O^*(r^{\alpha m})$. The Corollary follows from Theorem 22. \square

Remark 24. *While it would be nice to reduce the treewidth bound of a cubic graph from the $(1/9 + o(1))m$ of [FH06] to a simple $m/9$, any further reduction (e.g., to $m/10$) would result in no*

improvement in Corollaries 21 or 23, unless accompanied by improvements in some other aspect of the analysis.

While surprising, this fact is instantly obvious from the linear-programming results of Lemma 20. One interpretation is that it happens because, for $\alpha < 1/9$, the primal solution has weight 0 on the degree-3 III-reduction.

8. CONCLUSIONS

As noted in the Introduction, linear programming is key to our algorithm design as well as the analysis. We begin with a collection of reductions, and a preference order on them, guided by intuition. The preference order both excludes some cases (e.g., reducing on high-degree vertices first, we do not need to worry about a reduction vertex having a neighbor of larger degree) and determines an LP. Solving the LP pinpoints the “bad” reductions that determine the bound. We then try to ameliorate these cases: in the present paper we showed that each could be paired with another reduction to give a less bad combined reduction, but we might also have taken some other course such as changing the preference order to eliminate bad reductions. Using the LP as a black box is a convenient way to engage in this cycle of algorithm analysis and improvement, an approach that should be applicable to other problems.

While we focus on the linear program as a way to bound our key parameter, a graph’s III-reduction depth, Section 7 shows that it also applies to treewidth. Sharper results for (constraint) graphs of maximum degree 4 can be obtained simply by pruning down the LP.

Because the LP’s dual solution can be interpreted as a set of weights on edges and vertices of various degrees, the LP method introduced in [SS03], and further developed here, is closely connected to a potential-function approach. The determination of optimal weights can always be expressed as an optimization problem (see Eppstein’s [Epp04] and the survey [FGK05]), but its expression as an LP seems limited to cases where the CSP reductions are “symmetric” in the sense that they yield a single reduced graph. (A natural independent-set reduction is not symmetric in this sense, as reducing on a vertex v yields two reduced instances with different graphs: one deleting only the vertex v , the other also deleting all v ’s neighbors.) However, it can still be possible to plug bounds derived from asymmetric reductions into the LP method; for example the hypothesized algorithm in Theorem 22 might depend on asymmetric reductions. When the LP method is applicable, provably optimal weights are efficiently obtainable. Linear programming also provides an elegant framework and points the way to structural results like Lemma 20, but similar results could also be obtained under weaker conditions, outside the LP framework. For example, to prove Lemma 20, convexity of the solution space and linearity of the objective function would have sufficed.

It must be emphasized that the improvement of the present $19m/100$ depth bound over the previous $m/5$ is not a matter of a more detailed case analysis; indeed there are far fewer cases here than in most reduction-based CSP algorithms. Ultimately, the improvement comes from exploiting the constraint graph’s division into components. While this is very natural, its use in combination with the reduction approach and LP analysis is slightly tricky, and appears to be novel.

Linear programming aside, our approach seems not to extend to 3-variable CSPs, since a II-reduction would combine two 3-variable clauses into a 4-variable clause.

The improvement from $m/5$ to $19m/100$ is significant in that $m/6$ appears to be a natural barrier: In a random cubic graph, a III-reduction results in the deletion of 6 edges and a new cubic graph, and to beat $m/6$ requires either distinguishing the new graph from random cubic, or targeting a set of III-reductions to divide the graph into components. Such an approach would require new ideas outside the scope of the local properties we consider here.

Finally, we remark that it would be interesting to analyze further the behavior of algorithms on random instances. For example, it is shown in [SS06c] that for any $c \leq 1$, *any* Max 2-CSP instance

with constraint graph $G \in \mathcal{G}(n, p)$ can be solved in *linear expected time*. (Note that this is much stronger than succeeding in linear time *with high probability*.) Could this be extended to other problems? Could $2^{o(n)}$ runtime bounds be proved for random instances of problems such as Max Cut and Max 2-Sat with cn clauses, where $c \gg 1$? What about approximation results?

ACKNOWLEDGMENTS

We thank an anonymous referee and Daniel Raible for helpful comments. The first author's research was supported in part by EPSRC grant GR/S26323/01. We are also grateful to the LMS for supporting two research visits.

REFERENCES

- [AHe94] S. Arnborg, S.T. Hedetniemi, and A. Proskurowski (editors), *Special issue on efficient algorithms and partial k -trees*, Discrete Applied Mathematics **54** (1994), no. 2-3.
- [AKS87] N. Alon, J. Kahn, and P. D. Seymour, *Large induced degenerate subgraphs*, Graphs Combin. **3** (1987), no. 3, 203–211. MR MR903609 (88i:05104)
- [AP89] S. Arnborg and A. Proskurowski, *Linear time algorithms for NP-hard problems restricted to partial k -trees*, Discrete Applied Mathematics **23** (1989), no. 1, 11–24.
- [BE05] Richard Beigel and David Eppstein, *3-coloring in time $O(1.3289^n)$* , J. Algorithms **54** (2005), no. 2, 168–204.
- [Cre95] Nadia Creignou, *A dichotomy theorem for maximum generalized satisfiability problems*, J. Comput. System Sci. **51** (1995), no. 3, 511–522, 24th Annual ACM Symposium on the Theory of Computing (Victoria, BC, 1992). MR MR1368916 (97a:68076)
- [DCKP] F. Della Croce, Marcin J. Kaminski, and Vangelis Th. Paschos, *An exact algorithm for MAX-CUT in sparse graphs*, Operations Research Letters, to appear (available online 2006, doi:10.1016/j.orl.2006.04.001).
- [DJ02] Vilhelm Dahllöf and Peter Jonsson, *An algorithm for counting maximum weighted independent sets and its applications*, Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), January 2002.
- [DJW05] Vilhelm Dahllöf, Peter Jonsson, and Magnus Wahlström, *Counting models for 2SAT and 3SAT formulae*, Theoret. Comput. Sci. **332** (2005), no. 1-3, 265–291. MR MR2122506 (2005j:68055)
- [DP87] R. Dechter and J. Pearl, *Network-based heuristics for constraint-satisfaction problems*, Artif. Intell. **34** (1987), no. 1, 1–38.
- [DP89] Rina Dechter and Judea Pearl, *Tree clustering for constraint networks (research note)*, Artif. Intell. **38** (1989), no. 3, 353–366.
- [Epp04] David Eppstein, *Quasiconvex analysis of backtracking algorithms*, Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (New Orleans, LA, 2004) (New York), ACM, 2004, pp. 781–790.
- [FGK05] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch, *Some new techniques in design and analysis of exact (exponential) algorithms*, September 2005.
- [FH06] Fedor V. Fomin and Kjartan Høie, *Pathwidth of cubic graphs and exact algorithms*, Inform. Process. Lett. **97** (2006), no. 5, 191–196. MR MR2195217 (2006g:05199)
- [FK05] Martin Fürer and Shiva Prasad Kasiviswanathan, *Algorithms for counting 2-SAT solutions and colorings with applications*, Tech. Report TR05-033, Electronic Colloquium on Computational Complexity, March 2005, See <http://www.eccc.uni-trier.de/eccc/>.
- [FK07] Martin Fürer and Shiva P. Kasiviswanathan, *Exact Max 2-SAT: Easier and faster*, Proceedings of SOFSEM 2007, LNCS 4362, Springer, 2007.
- [GHN03] Jens Gramm, Edward A. Hirsch, Rolf Niedermeier, and Peter Rossmanith, *Worst-case upper bounds for MAX-2-SAT with an application to MAX-CUT*, Discrete Appl. Math. **130** (2003), no. 2, 139–155. MR MR2014655 (2004j:68077)
- [Hir00] Edward A. Hirsch, *A new algorithm for MAX-2-SAT*, STACS 2000 (Lille), Lecture Notes in Comput. Sci., vol. 1770, Springer, Berlin, 2000, pp. 65–73.
- [JKLS05] Klaus Jansen, Marek Karpinski, Andrzej Lingas, and Eike Seidel, *Polynomial time approximation schemes for max-bisection on planar and geometric graphs*, SIAM J. Comput. **35** (2005), no. 1, 110–119 (electronic). MR MR2178800 (2006f:68143)
- [KF02] Alexander S. Kulikov and Sergey S. Fedin, *Solution of the maximum cut problem in time $2^{|E|/4}$* , Zap. Nauchn. Sem. S.-Peterburg. Otdel. Mat. Inst. Steklov. (POMI) **293** (2002), no. Teor. Slozhn. Vychisl. 7, 129–138, 183.

- [KK06] A. Kojevnikov and A. S. Kulikov, *A new approach to proving upper bounds for MAX-2-SAT*, Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (Miami, FL, 2006) (New York), ACM, 2006, pp. 11–17.
- [KMRR05] Joachim Kneis, Daniel Mölle, Stefan Richter, and Peter Rossmanith, *Algorithms based on the treewidth of sparse graphs*, Graph-theoretic concepts in computer science, Lecture Notes in Comput. Sci., vol. 3787, Springer, Berlin, 2005, pp. 385–396. MR MR2213887
- [KR05] Joachim Kneis and Peter Rossmanith, *A new satisfiability algorithm with applications to Max-Cut*, Tech. Report AIB-2005-08, Department of Computer Science, RWTH Aachen, 2005.
- [KSTW01] Sanjeev Khanna, Madhu Sudan, Luca Trevisan, and David P. Williamson, *The approximability of constraint satisfaction problems*, SIAM J. Comput. **30** (2001), no. 6, 1863–1920 (electronic). MR MR1856561 (2002k:68058)
- [Mar04] Dániel Marx, *Parameterized complexity of constraint satisfaction problems*, Proceedings of the 19th IEEE Annual Conference on Computational Complexity (CCC'04), 2004, pp. 139–149.
- [MP06] Burkhard Monien and Robert Preis, *Upper bounds on the bisection width of 3- and 4-regular graphs*, J. Discrete Algorithms **4** (2006), no. 3, 475–498. MR MR2258338
- [NR00] Rolf Niedermeier and Peter Rossmanith, *New upper bounds for maximum satisfiability*, J. Algorithms **36** (2000), no. 1, 63–88.
- [Sch99] Uwe Schöningh, *A probabilistic algorithm for k-SAT and constraint satisfaction problems*, 40th Annual Symposium on Foundations of Computer Science (New York, 1999), IEEE Computer Soc., Los Alamitos, CA, 1999, pp. 410–414. MR MR1917579
- [SS03] Alexander D. Scott and Gregory B. Sorkin, *Faster algorithms for MAX CUT and MAX CSP, with polynomial expected time for sparse instances*, Proc. 7th International Workshop on Randomization and Approximation Techniques in Computer Science, RANDOM 2003, Lecture Notes in Comput. Sci., vol. 2764, Springer, August 2003, pp. 382–395.
- [SS04] ———, *A faster exponential-time algorithm for Max 2-Sat, Max Cut, and Max k-Cut*, Tech. Report RC23456 (W0412-001), IBM Research Report, December 2004, See <http://domino.research.ibm.com/library/cyberdig.nsf>.
- [SS06a] ———, *Generalized constraint satisfaction problems*, Tech. Report cs:DM/0604079v1, arxiv.org, April 2006, See <http://arxiv.org/abs/cs.DM/0604079>.
- [SS06b] ———, *An LP-designed algorithm for constraint satisfaction*, Proc. 14th Annual European Symposium on Algorithms, ESA, (Zürich, Switzerland, 2006), Lecture Notes in Comput. Sci., vol. 4168, Springer, September 2006, pp. 588–599.
- [SS06c] ———, *Solving sparse random instances of Max Cut and Max 2-CSP in linear expected time*, Comb. Probab. Comput. **15** (2006), no. 1-2, 281–315.
- [SS07] ———, *Polynomial constraint satisfaction: A framework for counting and sampling CSPs and other problems*, Tech. Report cs:DM/0604079v2, arxiv.org, February 2007, See <http://arxiv.org/abs/cs.DM/0604079>.
- [TSSW00] Luca Trevisan, Gregory B. Sorkin, Madhu Sudan, and David P. Williamson, *Gadgets, approximation, and linear programming*, SIAM J. Comput. **29** (2000), no. 6, 2074–2097.
- [Wil04] Ryan Williams, *A new algorithm for optimal constraint satisfaction and its implications*, Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP), 2004.

(Alexander D. Scott) MATHEMATICAL INSTITUTE, UNIVERSITY OF OXFORD, 24-29 ST GILES', OXFORD OX1 3LB, UK

E-mail address: scott@maths.ox.ac.uk

(Gregory B. Sorkin) DEPARTMENT OF MATHEMATICAL SCIENCES, IBM T.J. WATSON RESEARCH CENTER, YORK-TOWN HEIGHTS NY 10598, USA

E-mail address: sorkin@watson.ibm.com